



ENHANCED VISUAL USER INTERFACE
SUPPORT FOR DOMAIN-ORIENTED
APPLICATION COMPOSITION SYSTEMS

THESIS

Richard A. Guinto
Captain, USAF

AFIT/GCS/ENG/94D-06

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

19941228 116

AFIT/GCS/ENG/94D-06

Accession For	
NTIS GPO	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution	
Availability Codes	
Dist	Avail and/or Special
A-1	

JAN 24 1995

ENHANCED VISUAL USER INTERFACE
SUPPORT FOR DOMAIN-ORIENTED
APPLICATION COMPOSITION SYSTEMS

THESIS
Richard A. Guinto
Captain, USAF

AFIT/GCS/ENG/94D-06

DTIC QUALITY INSPECTED 2

Approved for public release; distribution unlimited

AFIT/GCS/ENG/94D-06

Enhanced Visual User Interface
Support for Domain-Oriented
Application Composition Systems

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Richard A. Guinto, B.S.C.S.
Captain, USAF

December 13, 1994

Approved for public release; distribution unlimited

Acknowledgements

I wish to thank my thesis advisor, Major Paul Bailor, for his patience and guidance throughout this research effort. I also want to thank my committee members, Dr. Thomas Hartrum and Lt Col Patricia Lawlis for their help and advice.

I'd also like to express my thanks to my fellow KBSE researchers for all the support and cooperation they gave me during our research.

Finally, I want to thank my children, Desiree and Daniel, for understanding that I had to spend a lot of time at school. Lastly, and most especially, I want to thank my wife, Leslie, for understanding, supporting, and most of all tolerating me during this period. Her patience and encouragement kept me going through my difficulties and successes.

Richard A. Guinto

Table of Contents

	Page
Acknowledgements	ii
List of Figures	vii
List of Tables	ix
Abstract	x
 I. Introduction	 1-1
1.1 Overview	1-1
1.2 Background	1-2
1.3 Problem	1-5
1.4 Assumptions	1-6
1.5 Sequence of Presentation	1-7
1.6 Summary	1-7
 II. Literature Review	 2-1
2.1 Introduction	2-1
2.2 Human-Computer Interaction	2-1
2.2.1 Usability	2-1
2.3 Human Factors	2-2
2.3.1 Design Guidelines	2-4
2.4 User Interface Evaluation Techniques	2-5
2.4.1 Heuristic Evaluation	2-5
2.4.2 Usability Testing	2-7
2.4.3 Guidelines	2-7
2.4.4 Cognitive Walk-through	2-8

	Page
2.4.5 Conclusion	2-9
2.5 Summary	2-9
III. Specification of System Enhancements	3-1
3.1 Introduction	3-1
3.2 Application Definition Capabilities	3-1
3.2.1 Defining the Application	3-3
3.2.2 Identifying components	3-5
3.2.3 Connecting the imports and exports	3-8
3.2.4 Specifying the update algorithm	3-11
3.2.5 Checking semantics, executing the application, and saving the application	3-14
3.3 Application Executive Visual Capabilities	3-16
3.3.1 Application Executive Metaphor Set	3-16
3.3.2 Application Executive Event Queue	3-18
3.4 Summary	3-19
IV. Design and Implementation of AVSI III	4-1
4.1 Introduction	4-1
4.2 Design Approach	4-1
4.2.1 Level of Effort	4-2
4.3 Implementation	4-2
4.3.1 Create/Edit Application	4-2
4.3.2 Create Subsystems/Build Imports-Exports	4-7
4.3.3 Building Application/Subsystem Update Algorithms	4-10
4.3.4 Application Executive Visualization	4-17
4.4 Conclusion	4-18

	Page
V. Testing and Validation of AVSI III	5-1
5.1 Introduction	5-1
5.2 Validation Domains	5-1
5.3 Testing	5-2
5.3.1 Compatibility Testing	5-2
5.3.2 Measuring Usability Improvement	5-2
5.3.3 Digital Logic Circuits Domain	5-3
5.3.4 Digital Signal Processing Domain	5-6
5.3.5 Event-Driven Circuits Domain	5-10
5.3.6 Cruise-Missile Domain	5-11
5.4 Analysis	5-14
5.5 AVSI III's Shortcomings	5-14
5.6 Conclusion	5-16
VI. Conclusion and Recommendations	6-1
6.1 Overview	6-1
6.2 Results of This Research	6-1
6.3 Recommendations for Future Research	6-2
6.4 Final Comments	6-3
Appendix A. Sample Session for AVSI III	A-1
A.1 Starting AVSI	A-1
A.2 Create a New Application	A-3
A.3 Edit the Application	A-3
A.3.1 To add a controlling subsystem-obj to the application	A-3
A.4 Edit the Subsystems	A-6
A.4.1 To add the primitive objects	A-6
A.4.2 To connect DRIVER's Imports and Exports	A-8

	Page
A.4.3 To create the application-obj's update-algorithm . .	A-12
A.4.4 To build DRIVER's Update Algorithm	A-14
A.4.5 To connect BCD-EXCESS3's Imports and Exports .	A-17
A.4.6 To build BCD-EXCESS3's Update Algorithm	A-20
A.5 Perform Semantic Checks	A-21
A.6 Execute the Application	A-23
Appendix B. REFINE TM Code Listings for AVSI III	B-1
Bibliography	BIB-1
Vita	VITA-1

List of Figures

Figure	Page
1.1. Domain-Oriented Application Composition Environment	1-2
1.2. Object Connection Update (OCU) Model	1-4
1.3. Architect with AVSI	1-5
3.1. Control Panel for AVSI II	3-4
3.2. Object Connect Update Window	3-6
3.3. System Composition Window and Technology Base Window	3-7
3.4. Imports/Exports Window and Technology Base Window	3-10
3.5. Update Algorithm Window	3-12
3.6. Application Executive Event Queue	3-15
3.7. Application Executive Metaphor Set	3-17
5.1. Exclusive-OR Circuit and Truth Table	5-3
5.2. Half Subtractor Circuit and Truth Table	5-4
5.3. Binary Array Multiplier Circuit and Truth Table	5-5
5.4. BCD to Excess-3 Decoder Circuit and Truth Table	5-6
5.5. Signal-With-Noise DSP	5-7
5.6. Window Demo	5-8
5.7. Four Sum Moving Average	5-9
5.8. Half Adder Event-Driven Circuit	5-10
5.9. Nand Gate Event-Driven Circuit	5-11
5.10. JK Flip Flop Event-Driven Circuit	5-12
5.11. Cruise Missile	5-13
A.1. BCD to Excess-3 Decoder Circuit	A-1
A.2. AVSI III Control Panel	A-2

Figure	Page
A.3. System Composition Window and Technology Base Window	A-4
A.4. System-Composition-Window	A-5
A.5. System Composition-Window	A-5
A.6. Edit-Subsystem-Window	A-7
A.7. DRIVER's Import-Export Window	A-9
A.8. Repositioned Icons	A-10
A.9. Import Area for BCD-EXCESS3	A-10
A.10. DRIVER's Import-Export Window	A-12
A.11. Import-Export MSP-Windows	A-13
A.12. Edit-Update-Algorithm Windows	A-15
A.13. Edit-Update-Algorithm	A-16
A.14. DRIVER's Update Algorithm	A-17
A.15. BCD-EXCESS3's Import-Export Window	A-18
A.16. BCD-EXCESS3's Import-Export Window	A-19
A.17. BCD-EXCESS3's Import-Export Window	A-20
A.18. BCD-EXCESS3's Update Algorithm	A-21
A.19. BCD-XS3 Application	A-25

List of Tables

Table	Page
2.1. Usability Principles	2-6
3.1. Execution Modes in Architect	3-2
3.2. Icon Operations Menu	3-9
3.3. Imports/Exports Icon Menu	3-9
3.4. Window Operations Menu	3-11
5.1. Exclusive-OR Test Results	5-4
5.2. Half Subtractor Test Results	5-5
5.3. Binary Array Multiplier Test Results	5-5
5.4. BCD to Excess-3 Decoder Test Results	5-7
5.5. Signal-With-Noise Test Results	5-8
5.6. Window-Demo Test Results	5-8
5.7. Four-Sum-Moving-Average Test Results	5-9
5.8. Half Adder Event-Driven Test Results	5-10
5.9. Nand Gate Event-Driven Test Results	5-11
5.10. JK Flip Flop Event-Driven Test Results	5-12
5.11. Cruise Missile Flight Test Results	5-13
5.12. Usability Improvement	5-14
A.1. BCD to Excess-3 Decoder Truth Table	A-23

Abstract

This research refined the functionality and usability of a previously developed visual interface for a domain-oriented application composition system. The refinements incorporated more sophisticated user interface design concepts to reduce user workload. User workload was reduced through window reordering, menu redesign, and Human Computer Interaction techniques such as; combining repetitive procedures into single commands, reusing composition information whenever possible and deriving new information from existing information. The Software Refinery environment, including its visual interface tool INTERVISTA, was used to develop techniques for visualizing and manipulating objects contained in a formal knowledge base of objects. The interface was formally validated with digital logic-circuits, digital signal processing, event-driven logic-circuits, and cruise-missile domains. A comparative analysis of the application composition process with the previous visual interface was conducted to quantify the workload reduction realized by the new interface. Level of effort was measured as the number of user interactions (mouse or keyboard) required to compose an application. On average, application composition effort was reduced 34.0% for the test cases.

Enhanced Visual User Interface Support for Domain-Oriented Application Composition Systems

I. Introduction

1.1 Overview

As computers are used more in the workplace, the need for developing usable computer hardware and software products becomes more important. Users are offered more complex data and functions. To allow users to take advantage of these advanced systems, software developers must provide more sophisticated visually-based user interfaces.

Architect, a prototype domain-oriented application composition and generation system, is the subject of ongoing research by the Knowledge-Based Software Engineering (KBSE) research group at the Air Force Institute of Technology (AFIT). Originally developed by Anderson and Randour (1, 16), it was later enhanced with a visual user interface. This interface, called Architect Visual System Interface (AVSI), gives a graphical representation of the domain-specific language and allows generating, viewing, and manipulating application specific information (23). Further refinement of AVSI extended Architect's usability by incorporating sophisticated graphics and improving the application development process resulting in AVSI II(3). A significant amount of research still remains for the further refinement of the interface. This thesis will briefly present the history of Architect

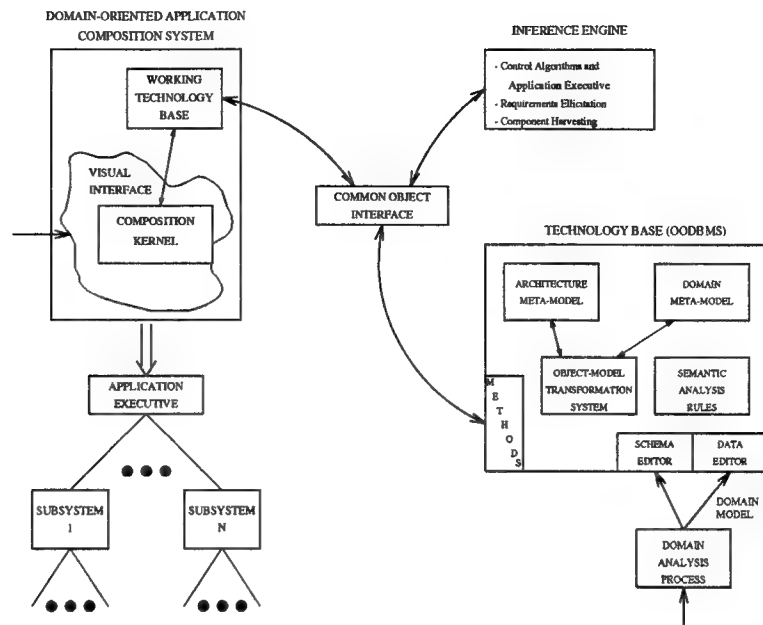


Figure 1.1 Domain-Oriented Application Composition Environment

and AVSI and the specific challenges involved in improving Architect's interface, taking Architect to its highest level of usability.

1.2 Background

The KBSE research group at AFIT conducts research into the various aspects of domain-oriented application composition and generation as shown in Figure 1.1. Architect runs within the Software RefineryTM development environment. It is centered on a Lisp-based wide-spectrum specification language called REFINE. Architect was designed to be used by an application specialist. The application specialist is familiar with the overall domain and understands what the new application must do to meet system requirements and specifications (8:4). Architect provides the application specialist with the tools to compose applications within specific domains.

The structure of an Architect application is based on the Object Connection Update (OCU) Model developed by the Software Engineering Institute (SEI) (9). The OCU model consists of several software elements used to express a design in the form of subsystems and hardware interfaces. A subsystem, visualized in Figure 1.2, consists of a controller, a set of objects, an import area, and an export area. A software system, or application, is described as a set of subsystems controlled by an executive function. An executive, is a high level supervisory subsystem that coordinates the behavior and information flow between subordinate subsystems. The OCU model treats subsystems as self-contained, abstract units; however, there are no restrictions against a subsystem acting as an object controlled by another subsystem. More complete definitions of the subsystem components are provided below.

- **Controller:** The controller aggregates a set of objects and manages the connections between them and the flow of information to and from them. Controllers contain an update function that defines the functionality (or mission) of the subsystem.
- **Import area:** The import area is the central collection point for the inputs needed by the objects in a subsystem. The import area collects data from other subsystems in an application and makes it available to the objects.
- **Export area:** The export area is the central distribution point for the outputs of a subsystem that is needed by other subsystems in an application.
- **Object:** An object models the behavior of real-world components. An object maintains a state and has an associated set of algorithms responsible for transforming input data into the state data.

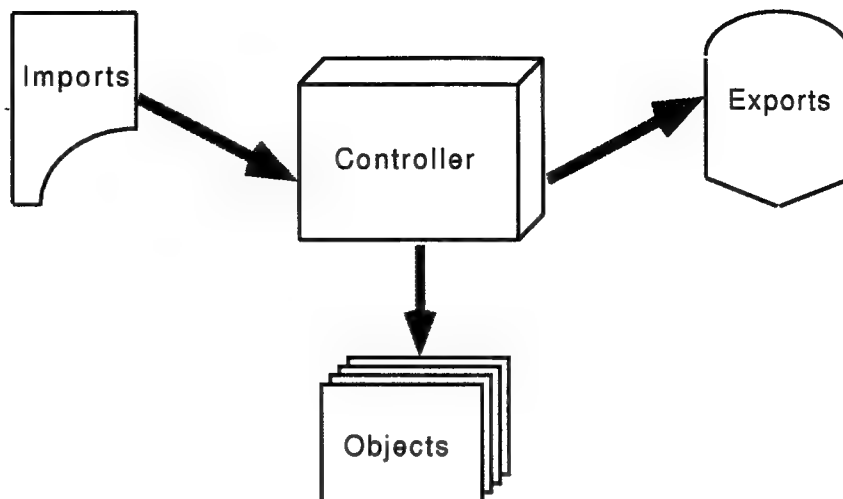


Figure 1.2 Object Connection Update (OCU) Model

The application specialist manipulates the domain model to develop a preliminary software specification within Architect. This specification can then be compiled, and the resulting code can be executed to simulate system operation. The execution is monitored for desired functionality of the requested system. The development/compilation/execution cycle can be repeated until the required behavior is achieved. Translation of the specification into a deliverable software application in Ada, C, or another language, can then be accomplished.

AVSI was developed to replace Architect's rudimentary command line interface (23). It allows the application specialist to visualize a "mental model" (11:100) of the application on the computer monitor rather than having to mentally visualize it. Figure 1.3 shows the relationship between AVSI and the components of Architect. AVSI gives the application specialist the basic functional capability of a visual user interface. A major improvement to AVSI, called AVSI II, provided the application specialist with a more mature version of a visual user interface (3). AVSI II extended Architect's usability by incorporating

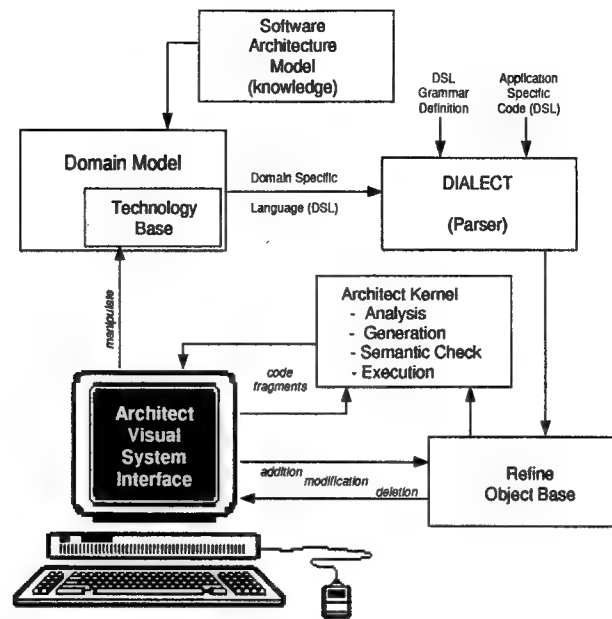


Figure 1.3 Architect with AVSI

sophisticated graphics, adding basic visual interface support for an Application Executive, and improving the application development process. Nevertheless, Architect combined with AVSI II still lacked the high degree of functionality desired to minimize the time and complexity to design a software system. Therefore, the goal of this thesis endeavor was to continue functional development of the Architect user interface, and further reduce the level of effort required to enhance usability of the system.

1.3 Problem

In order to make Architect even more “user friendly” for the application specialist, AVSI II needed additional enhancements. AVSI II provided an application specialist with a broad base of functionality, but some of the processes were still repetitive, time consuming,

or tedious. The challenge was to make AVSI III even more usable. To meet the challenges described above, the following research goals and objectives were established:

1. To extend support to infer controller update algorithms by synthesizing a sequencing algorithm from the visual representation of control data. The current implementation of AVSI II requires the application specialist to explicitly define the execution order of the subsystem(s) and primitive(s). An improved implementation would infer an update algorithm using as a basis the import and export (i.e., data) connections of the subsystem(s) and primitive(s) of the application.
2. To investigate and implement an appropriate visualization method to provide visual support of the application execution. AVSI II supported application execution by displaying a textual representation, in the form of an event queue, of the initial and final states of the application. The enhanced event queue would display not only the initial and final states, but also display the intermediate states.
3. To support the visualization needs for Architect to compose applications using object from multiple domains.
4. To test the new interface and quantify AVSI III's improvements.

1.4 Assumptions

This research will make several basic assumptions. The domains used to validate AVSI III, the digital logic-circuits, event-driven digital logic-circuits, digital signal processing circuits, and cruise missile domains are assumed to be adequate for testing and validation of AVSI III. These domains have been validated by previous work and would

provide sufficient coverage over a couple of execution modes. The execution modes currently supported include non-event-driven-sequential, event-driven-sequential, and time-driven sequential.

1.5 Sequence of Presentation

The remainder of this thesis is organized as follows:

1. Chapter 2 contains a review of current literature concerning human-computer interactions, human factors, and user interface evaluation techniques.
2. Chapter 3 includes a description of the overall operational concept of AVSI III and its relationship to Architect. It also describes an analysis of AVSI III and the changes implemented. This chapter also details the visualization of the application executive, the addition of an automatic update algorithm mechanism, the redesign of the import/export connection process, and the quantification of Architect's changes.
3. Chapter 4 presents AVSI III's design, design tradeoffs, and implementation details.
4. Chapter 5 addresses the testing and validation of AVSI III and presents an analysis of its capabilities.
5. Chapter 6 contains conclusions about the AVSI III research effort as well as including recommendations for future research.

1.6 Summary

The research proposed in this thesis is essential to meet the high demands an application specialist places on Architect. The importance of the role of application composition

and generation systems like Architect is increasing within the software industry(13). The new visual user interface implementation, AVSI III, provides the application specialist with the high degree of functionality required to design complex and sophisticated software systems.

II. Literature Review

2.1 Introduction

This chapter summarizes current research in the areas of Human-Computer interaction, human factors, data-flow analysis techniques, dependency graphs, and user interface evaluation techniques. In this chapter, a brief description of Human Computer Interaction concepts is presented. In addition, the goals of human factors and their evaluation criteria are discussed. Finally, four different user interface evaluation techniques are summarized.

2.2 Human-Computer Interaction

Human-Computer Interaction (HCI) is what happens when a human and a computer system interact to perform tasks. This field is new and is dedicated to determining how best to make this interaction work. HCI includes user interface hardware and software, user and system modeling, cognitive and behavioral science, human factors, empirical studies, methodology, techniques, and tools (6). The goal in HCI is to provide the user with a high degree of usability.

2.2.1 Usability. The main focus of usability is to improve user performance of tasks with an interactive computer system. Usability is a combination of the following user-oriented characteristics (18):

- Ease of learning
- High speed of user task performance
- Low user error rate

- Subjective user satisfaction
- User retention over time

2.3 Human Factors

The goal of human factors is to optimize human performance, including error reduction, increased throughput, user satisfaction, and user comfort (6). User interaction designers are realizing that the user should not have to adapt to the interface, but rather the interface should be designed so that it is intuitive and natural for the user to use. Several factors weigh heavily into what human factors are:

- **User interaction standards** – are documents that give requirements for user interaction design. One such document for user interaction is the *User-Computer Interface*, MIL-STD-1472C, revised (1990). However, sometimes they require intense interpretation and tailoring to be useful in user interaction design. Their main advantage is that they draw attention to the user interface, but they are generally too vague to offer effective guidance (6).
- **User interaction design guidelines** – are general in their applicability and require a fair amount of interpretation to be useful. Guidelines are mostly educated opinions based on experience. One of the best known examples of a collection of guidelines is the technical report *Guidelines for Designing User Interface Software* (19). A guideline's main advantage is that it offers flexible guidance and helps establish design goals and decisions, but it must be tailored to produce specific design rules. The main difference between standards and guidelines is that standards are enforceable in user

interface, while guidelines are merely suggestions as to how to produce a good user interface.

- **Commercial style guides** – are documents produced by an organization or vendor that are made commercially available. They provide a much more concrete and useful framework for design than a standards document. A style guide typically includes the following:

1. Description of a specific interaction style or object, including both its *look* and *feel*.

2. Guidance on when and how to use a particular interaction style or object. Style guides can provide the basic conventions for a specific product or for a family of products. Their main advantage is that they improve consistency of the user interaction design. Examples of currently popular commercial style guides are:

- (a) *OSF/MotifTM* Style Guide from Quest Windows Corporation (15).

- (b) *OpenLookTM* from AT&T/Xerox/Sun

- **Customized style guides** – are related to commercial style guides in that they contain similar information with the exception that they also contain specific recommendations for various aspects of the user interaction design. These guidelines unlike commercial guides are made for specific products. Their main advantage is providing consistent, explicit, unambiguous information for a user interaction design. However, developing a customized style guide can require a large amount of resources.

2.3.1 Design Guidelines. The design guidelines described above are very important in producing a high-quality user interface. However, the number of guidelines available are numerous and an attempt to list all or even most of them is beyond the scope of this research. The following guidelines were found to be the most relevant in improving AVSI.

2.3.1.1 User-Centered Design. Practicing user-centered design emphasizes a user interface that is easy to use by the user, not what is easy for the developer to build. This implies that the interface designer know the user. He/she must know the particular characteristics that make up the type of users that will use the interface. This can not be done simply by observation or interviews. Methods such as user analysis, task analysis, information flow analysis, and time-and-motion analysis may be used to determine a better understanding of the characteristics of the users of the user interface.

Another aspect of user-centered design is based on the prevention of user errors. Good interaction design anticipates potential problem areas and helps the user to avoid making mistakes. For example, to prevent typographical errors the user interface should display, as choices to the user, a list of legitimate choices from which the user may choose.

2.3.1.2 Human Memory Issues. “Make the application memorable by reducing the user’s need to memorize.” (20:133) A human’s short-term memory must be taken into account when designing a user interface. The human capacity is normally measured as the famous “seven plus or minus two chunks” (12). This limitation can be overcome in user interface design by using identification instead of recollection. For example, identifying a choice from a menu is much easier for the user than having to recollect

all possible choices and then typing in one of them as a response. Preventing the user from having to memorize also decreases typographical errors.

2.3.1.3 Menu Issues. “Close targets are faster to acquire than far ones: Keeping everything but menu bars and other edge-hugging items close to the area of interest saves the user time.” (20:206) Reducing the distance when moving the mouse pointer or cursor increases usability of the user interface. This concept goes hand-in-hand with the concept that reducing or eliminating navigation also makes a better user interface.

Design menus to display only those options that are actually available in the current context. When it comes to displaying menus for user selection, limiting choices available to those that make “sense” at the time prevents potential errors and also decreases the memory load on the user. For example, clicking on an icon that represents a print task selectively displays commands meant specifically for the printer.

2.4 User Interface Evaluation Techniques

A brief search of evaluation techniques to identify the technique that should be used to evaluate AVSI was accomplished, and four techniques were discovered. These techniques (7) are heuristic evaluation, usability testing, guidelines, and cognitive walk-through. The good and bad points of each techniques are described below.

2.4.1 Heuristic Evaluation. User Interface (UI) specialists study the interface in depth and look for properties that they know, from experience, will lead to usability problems. Evaluators accomplish Heuristic evaluation (14) by looking at an interface and trying to come up with an opinion about what is good and bad about the interface. Such

Table 2.1 Usability Principles

Simple and natural dialogue
Speak the user's language
Minimize user memory load
Be consistent
Provide feedback
Provide clearly marked exits
Provide shortcuts
Good error messages
Prevent errors

an evaluation should be conducted using rules that are derived from certain usability guidelines such as the nine basic usability principles (14) listed in Table 2.1.

These principles correspond to what is considered to be important to the user interface community. Studies (14) have shown that the performance of every evaluator is not the same. While a “good” evaluator might be able to find most of the usability problems of an interface, a “poor” evaluator could sometimes find problems that were missed. Aggregates of evaluators are formed by having several evaluators conduct an individual heuristic evaluation. The individual results are then compiled together to form an aggregate evaluation. Between three and five evaluators are recommended to achieve a decent evaluation.

2.4.1.1 Advantages. Heuristic evaluation is very intuitive and easy to motivate people to perform. Therefore, minimum planning is required. Compared to the other methods, it identifies not only more problems, but more serious problems (14). It draws most of its strength from the skilled UI professionals who use it, and the heuristic evaluation method uses a minimum of resources.

2.4.1.2 Disadvantages. The disadvantages of using heuristic evaluation include biasing by the current mind set of evaluators, and it doesn't provide a solution to problems found or doesn't generate breakthroughs in the design. This method requires extensive UI expertise. To obtain a good evaluation, multiple evaluators are necessary.

2.4.2 Usability Testing. In usability testing the interface is studied under real-world or controlled conditions with evaluators gathering data on problems that arise during its use.

2.4.2.1 Advantages. It is very good at finding serious and recurring problems and at avoiding low-priority problems. Usability Testing is also good at finding serious problems.

2.4.2.2 Disadvantages. This method is very costly and requires massive amounts of time. It, like heuristic evaluation, requires UI expertise. Even though it is very costly, one would think that it would fair better than most methods, but it still fails to find all the serious problems and missed consistency problems (7).

2.4.3 Guidelines. Guidelines provide evaluators with specific recommendations about the design of an interface. The ESD/MITRE compilation of user interface design guidelines, *Guidelines for Designing User Interface Software* (19), contains 944 guidelines. This report represents the most comprehensive guidance available for designing user interface software. The report is organized into six functional areas of user-system interaction. These areas are: Data Entry, Data Display, Sequence Control, User Guidance, Data Transmission, and Data Protection. The individual guidelines contained in this report are

generally worded so that they may be applied across many different system applications. Because of this generalization, the guidelines must be translated into specific design rules before they can be used.

Because guidelines may conflict with each other, designers must pinpoint the importance of one guideline over another. Also, since there are so many guidelines to choose from, designers may have to limit their choices to only the most important ones. When a final cut is made, all designers have to abide by the chosen guidelines so as to ensure a consistent design. Finally, after the design is complete, it must be evaluated against the original design requirements to ensure all design rules have been followed.

2.4.3.1 Advantages. A guidelines evaluation is the best at finding recurring and general problems. This method works best when UI specialists are used as evaluators, but in a pinch, software developers can be used as a reasonable alternative.

2.4.3.2 Disadvantages. Using this approach, evaluators can miss a large number of severe problems and it is very time consuming (7).

2.4.4 Cognitive Walk-through. The cognitive walk-through method (10) combines software walkthroughs with a cognitive model of learning by exploration. The developers of an interface walk through the interface in the context of core tasks a typical user will need to accomplish. The designer specifies a series of core tasks that are deemed important. For each core task, the sequence of user actions is specified by the designer. The actions and feedback of the interface are compared to the user's goals and knowledge, and discrepancies

between the user's expectations and the steps required by the interface are noted. In essence, the designer is doing a hand simulation of the processes involved.

2.4.4.1 Advantages. The cognitive walk-through approach helps to define the user's goals and assumptions. This technique can be done by software developers.

2.4.4.2 Disadvantages. This method is tedious and sometimes requires too much detail, and the problems found with this method are typically less general and less recurring. Also, the amount of time required to analyze the tasks defined by this technique make it unattractive to most developers(10).

2.4.5 Conclusion. In conclusion, it appears that a Heuristic evaluation of AVSI II would be the most feasible. It can identify more serious problems than any of the four methods without expending a large amount of resources.

2.5 Summary

Even with all the research being done concerning human-computer interfaces and human factors, developing user interfaces still involves concentrating on the user. Designing user interfaces that are centered on the user produces a more usable user interface. With the emphasis on practicing user-centered design, the interface designer can develop a product that is easier to use and understand.

III. Specification of System Enhancements

3.1 Introduction

This chapter describes the current and proposed capabilities of Architect's Visual System Interface (AVSI) II. This description is broken down into two aspects, Application Definition and Application Executive Visualization. The general process used to analyze the current capabilities of AVSI II was based on determining the number of user actions required to complete some step. This helped to pinpoint the areas where improvements were necessary. Validation of the success of whether significant usability improvement were made was achieved by measuring user actions required for both versions of AVSI and making comparisons and conclusions.

3.2 Application Definition Capabilities

The baseline Architect version 2.0 gave the application specialist the ability to build domain-specific applications. Previously validated domains are: digital logic circuits (1), digital signal processing (22), event driven logic circuits (21), cruise missile (21), and application executives (24).

Applications built using AVSI II must be defined as executing in a specific execution mode. In the non-event-driven sequential mode of execution, simulation entities are updated in a fixed order during each execution. This fixed order in Architect is called the update algorithm. An event-driven sequential application executes as the result of executive service of events that are asynchronously raised by the application subsystems and the executive. A time-driven sequential application contains subsystems that react to changes

Table 3.1 Execution Modes in Architect

Digital Logic Circuits	Non-event-driven-sequential
Digital Signal Processing	Non-event-driven-sequential
Event-Driven Circuits	Event-driven-sequential
Cruise-Missile	Time-driven-sequential
Application Executive	Executive

in the clock. So, an update algorithm is not required when specifying an application in either the event-driven sequential and time-driven sequential execution mode. Table 3.1 shows, the domain appropriate for a specific execution mode.

To create an application definition, the application specialist uses a menu driven graphical user interface (GUI) to interact with Architect to perform operations on a domain-specific application. First, he must define an application. Defining an application is further refined into several steps. The necessary steps are:

1. Select a domain.
2. Select an appropriate execution mode.
3. Name the application.
4. Create and name a controlling subsystem.

With an application defined, any and all secondary subsystem(s) and/or primitive(s) must be identified, named, and linked to their controlling subsystem. Next, in the case of the non-event-driven sequential execution mode, an update algorithm, which defines the order in which each subsystem and/or primitive is updated must be defined. An update algorithm is not needed when developing applications that are executed in event-driven-sequential or time-driven-sequential modes. Next, the imports and exports of subsystem(s)

and/or primitive(s) must be "connected". When the previous steps have been accomplished, a semantic check is performed to ensure appropriate conditions are met. If any errors are found, corrections must be made before execution of the application can proceed. Execution of the application is then simulated, and the application's behavior can be compared with what was expected to determine if modifications are necessary.

AVSI II currently aides the application specialist in performing the necessary steps to create an application. AVSI II's central interface was based on the use of a control panel window as shown in Figure 3.1. This necessitates the user to maneuver through a hierarchy of menu selections before actually selecting the intended command. A goal of this research was to design and implement modifications to AVSI II to increase usability by decreasing the user's workload. A careful study of the current process identified several areas that needed to be improved. The following subsections describe AVSI II's design, and describes the modifications made to improve it. A stepwise approach was used to ensure a fully functional interface at all times during this research effort and to correctly construct an improved interface called AVSI III.

3.2.1 Defining the Application.

3.2.1.1 Description of the Current Application Definition Process. To create a new application, the application specialist must click on the "Create New Application" button on the control panel. The user is then prompted for a domain. AVSI II requires the user to choose a domain and an execution mode for the application being built. Non-event-driven-sequential mode is selected for the digital logic circuit and digital signal processing domains, while the Event-driven-sequential mode is selected for the

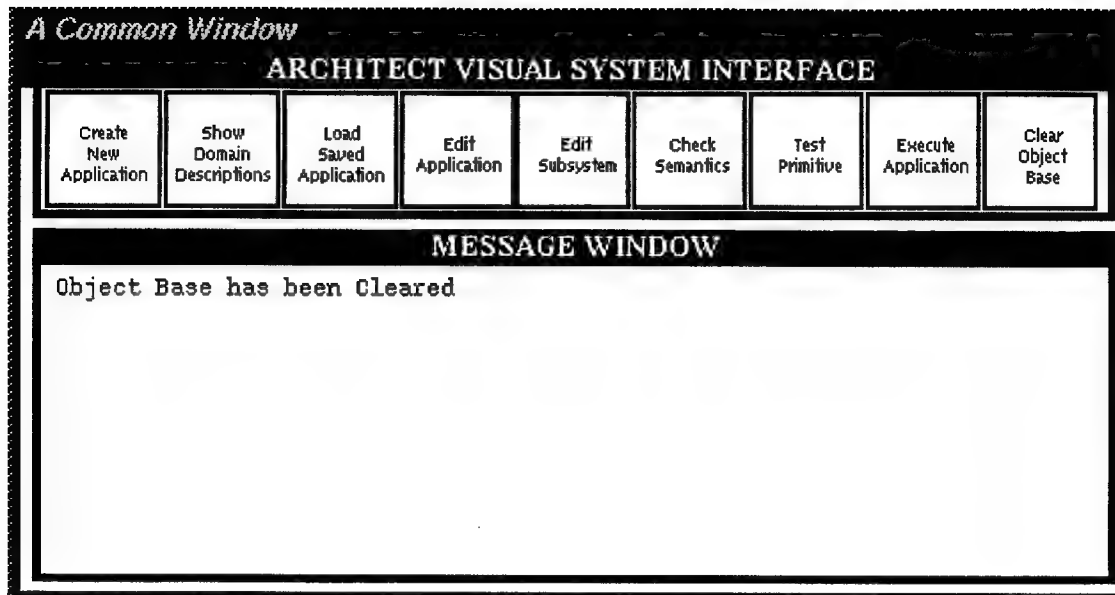


Figure 3.1 Control Panel for AVSI II

Event-driven circuit domain, and Time-driven-sequential mode is selected for the Cruise-Missile domain. Next, the user is prompted for an application name.

3.2.1.2 Analysis of the Application Definition Process. Examination of the current process indicates that the current operation requires the user to recall from memory what the execution mode of the application should be. Because the current design of Architect dictates strict adherence to the matching of domain with the exact execution mode (see table 3.1), this prerequisite can be implemented in a more fitting manner.

3.2.1.3 Improvements to the Application Definition Process. Improvements in AVSI III incorporate the capability to automatically select the execution mode required based on the chosen domain, thus reducing the amount of information the user needs to elaborate to Architect.

3.2.2 Identifying components.

3.2.2.1 Description of Current Component Identification Process. Once an application name is defined, AVSI II requires the user to choose "Edit Application" from the AVSI control panel. A submenu then displayed a list of loaded applications, prompting the user to choose an application to edit. After selecting the appropriate application name, another submenu is displayed giving the user a choice of either selecting "Edit Application Update", "Edit Application Components", "Save Application", or "Abort". "Edit Application Component" should be chosen, displaying a composition window while showing the user the control hierarchy relationships. Currently, there is only one icon with text labeling it an "APPLICATION-OBJ" with the name of the application as defined. Clicking on the blue background of the window brings up yet another submenu in which the user should choose "Create New Subsystem". The user is then prompted for the subsystem's name. Once the subsystem is named, an outline of the subsystem icon is displayed in the composition window. This outline can be positioned in the composition window as desired by the user. Once the icon is positioned, clicking a mouse button anchors that icon. The subsystem icon is then displayed with text labeling it an "SUBSYSTEM-OBJ" with the name defined by the user. Additional subsystems may be created in the same fashion. The procedure for linking subsystem(s) to its controlling subsystem is then accomplished. Once the subsystem is created and named, it is linked to the application by clicking the mouse on the subsystem icon and selecting "Link to Source" from the pop-up menu. The link is then created.

In AVSI II, creating primitives, was accomplished by first selecting “Edit Subsystem” and choosing the appropriate controlling subsystem. The Object Connection Update (OCU) model (9) is then displayed as shown in Figure 3.2. Clicking on the icon labeled “Objects” then displays two windows. The windows displayed are the “SUBSYSTEM-OBJ” window and “Technology Base” window. The application specialist then selects any primitive(s) that the application design requires from the “Technology Base Window” and “positions” it into the “SUBSYSTEM-OBJ” window.

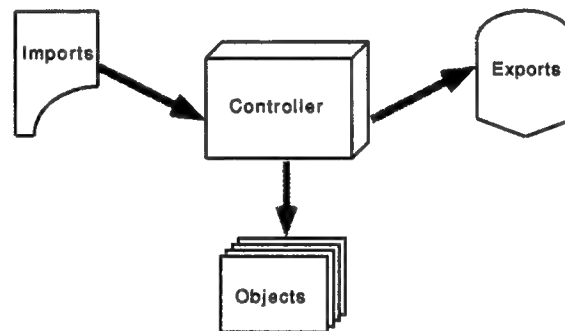


Figure 3.2 Object Connect Update Window

3.2.2.2 Analysis of Component Identification. Analyzing the processes required when identifying components, the necessary tasks to accomplish are:

1. Create subsystem(s).
2. Create primitive(s).
3. Linking subsystem(s) and/or primitive(s) to their controlling subsystem.

The two windows required to complete the steps mentioned are the “System Composition Window” and the “Technology Base Window”. So it is evident that AVSI III should

automatically display both the "System Composition Window" and the "Technology Base Window" as shown in Figure 3.3.

3.2.2.3 Improvements to Component Identification. In the "System Composition Window" the application specialist must create a controlling subsystem, and may create any secondary subsystem(s). The application specialist may also select any primitive(s) that the application design requires from the "Technology Base Window" and "position" it (the primitive) into the "System Composition Window". Linking of subsystem(s) and/or primitive(s) is identical to the previous version of AVSI II (3). This significantly reduces the effort required by the application specialist.

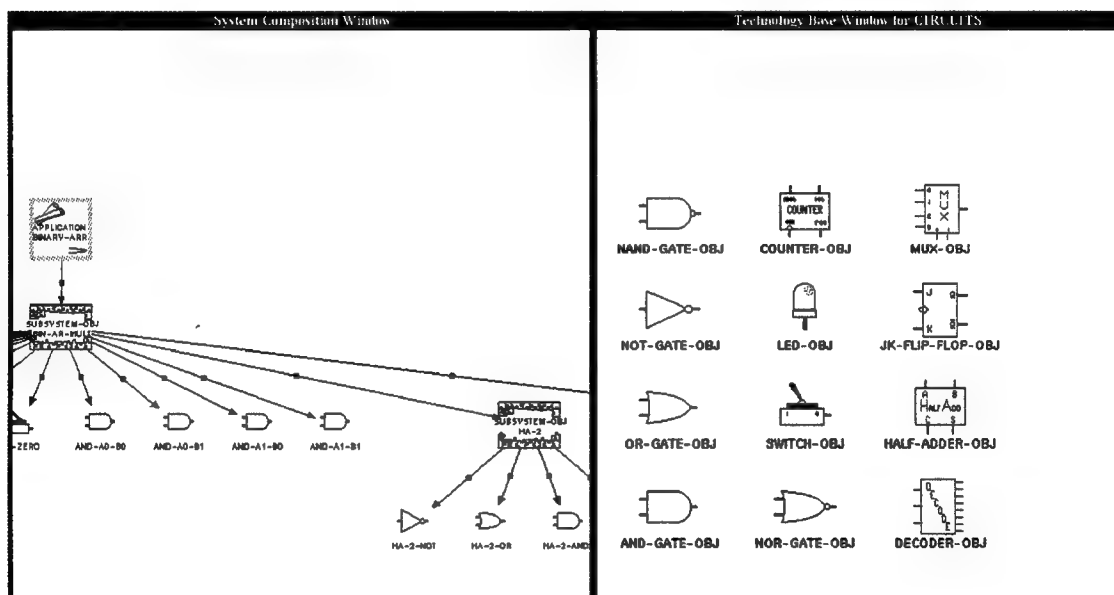


Figure 3.3 System Composition Window and Technology Base Window

3.2.3 Connecting the imports and exports.

3.2.3.1 Description of Current Imports/Exports Connection Process. In AVSI II, connecting the imports and exports of the application subsystem(s) and/or primitive(s), is accomplished by first selecting "Edit Subsystem" and choosing the appropriate subsystem. The Object Connection Update (OCU) model (9) is then displayed as shown in Figure 3.2. Clicking on the icon labeled "Import Area" then prompts the user to choose either "Make Connections", "View Information", or "Abort". Selecting "Make Connections" displays the "Imports/Exports" window. The connections are then made between the imports and exports of the appropriate subsystem(s) and/or primitive(s).

3.2.3.2 Analysis of Imports/Exports Connection. AVSI II requires the application specialist to navigate a series of menu structures in order to make the import/export connections. It required a minimum of five user actions before the appropriate windows are displayed to allow the user to make the needed connections. The connecting of imports and exports is made laborious by the hierarchical navigating of menus. Minimizing menu navigation will greatly reduce the level of effort required to accomplish this task.

3.2.3.3 Improvements to Imports/Exports Connection. In AVSI III, connecting the imports and exports is accomplished by use of the "Icon Operations" menu. Modifying the menu to include a selection to go directly to the "Imports/Exports" window reduces the level of effort. Clicking on any icon displays the "Icon Operations" menu giving

the user several choices. The choices given an application specialist are shown in Table 3.2.

Table 3.2 Icon Operations Menu

Delete-Object
Edit Label
Move
Link Multiple Targets
Link to Source
Make Connections
Edit Application Update
Pretty Print Object
View/Edit Object Attributes
View/Edit Object Descriptions
Mcn Icon
Mcn Object
Check Semantics
Execute Application
Save Application
Abort

Selecting "Make Connections" clears the screen of the underlying windows and displays the "Imports/Exports" window displaying the highest level subsystem of the application. Clicking on the subsystem icon brings up another menu in which "Make Internal Connections" should be chosen. The connections are then made as previously described. Using AVSI III, the application specialist only performs two user actions to navigate the appropriate windows before being able to make the connections required.

Table 3.3 Imports/Exports Icon Menu

Move Icon
Mcn Icon
Mcn Object

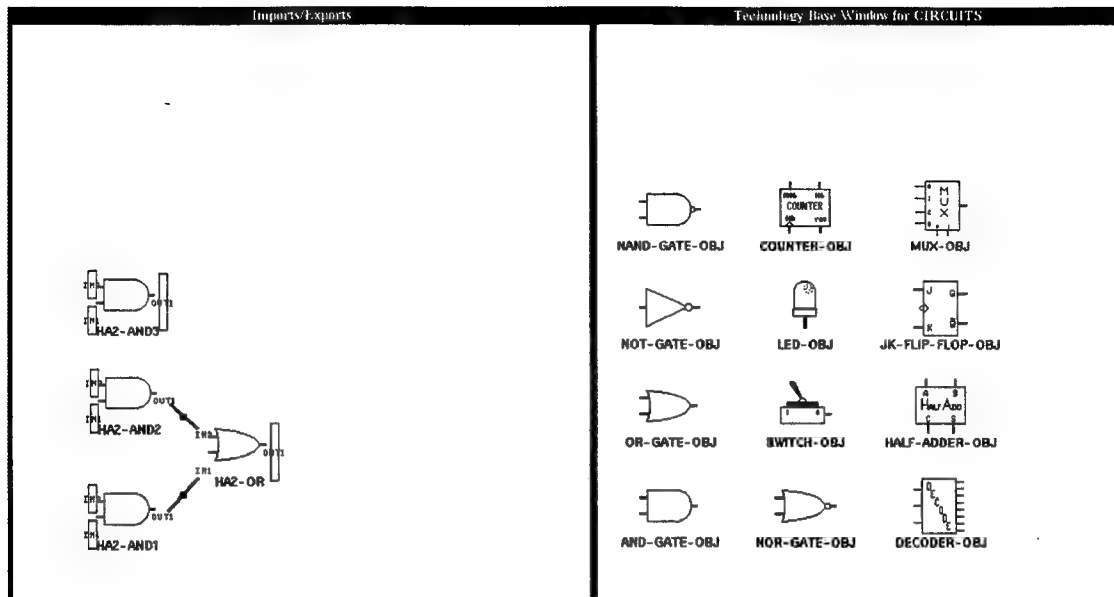


Figure 3.4 Imports/Exports Window and Technology Base Window

To further increase usability, the ability to add additional primitives is included. In fact, the entire subsystem composition process can be accomplished with this mode by incrementally selecting new primitives and connecting them together.

Clicking on an import icon and an export icon produces a connection between the two. When all connections are made, the “Imports/Exports” window is deactivated by using the “Window Operations” menu shown in Table 3.4.

Table 3.4 Window Operations Menu

Shrink
Deactivate
Move
Rescale Window
Clip Icon Labels
Restore Icon Labels
Refresh
McN Window
Abort

3.2.4 Specifying the update algorithm.

3.2.4.1 Description of the Current Update Algorithm Specification Process.

Once the imports and exports have been defined, in the case of the non-event-driven-sequential execution mode, the update algorithm for the application must be specified. AVSI II required the user to accomplish this in two parts. These two parts are defining the overall application update algorithm and defining each subsystem's update algorithm. In defining the application update algorithm, the user must select the "Edit Application" button in the control-panel. Next, the user must choose the application whose update algorithm is being defined. Then, selecting "Edit Application Update" from another pop-up submenu provides the appropriate diagram windows that are necessary to accomplish this task. In defining the update algorithm for a subsystem, the user must select the "Edit Subsystem" button in the control-panel. Next, the user must choose the subsystem whose update algorithm is being defined. The subsystem diagram window is displayed providing the user with the OCU model of the subsystem. Then, clicking on the "Controller" icon provides the appropriate diagram windows that are necessary to accomplish this task.

After the appropriate application/subsystem was selected, AVSI II presented three “Update Algorithm” windows: one giving a graphical (iconic) view of the update algorithm, one giving the textual view, and one called the “Controllee” window. The “Controllee” window displays a copy of the icons created in the “System Composition” window and two additional icons that represent if-then and do-while constructs. To build the update algorithm, the application specialist clicks the mouse on an icon in the “Controllee” window. The mouse cursor changes to a “bulls-eye” shape and the application specialist clicks on a “nub” in the “Edit Update Algorithm” window. AVSI II automatically adds the icon to the “Edit Update Algorithm” window and a textual representation to the textual update algorithm window. To create a six object update algorithm, the application specialist performed two mouse actions per object for a total of 12 actions.

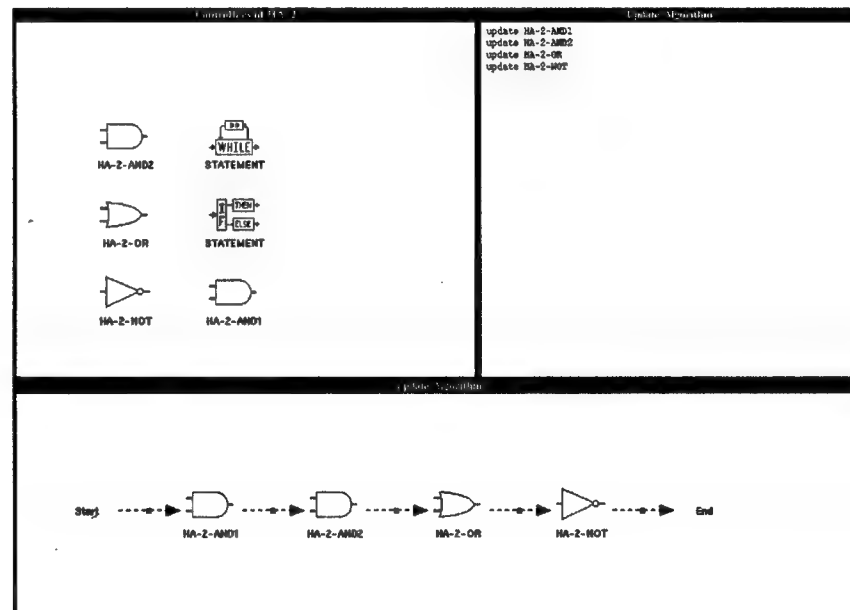


Figure 3.5 Update Algorithm Window

3.2.4.2 Analysis of the Update Algorithm Specification. The procedure for building the update algorithm under AVSI III needed to be less tedious. Using information from the import/export connections process, AVSI III can infer an update algorithm. This reduces the amount of user actions.

3.2.4.3 Improvements to the Update Algorithm Specification. Enhancements in AVSI III provide the user with the capability to build the update algorithms automatically or manually. AVSI III requires the user to do this in two parts. Starting from the "System Composition", window the user clicks on an icon producing the "Icon Operations" menu and selects "Edit Application Update". AVSI III then displays the "Update Algorithm" windows and determines which update algorithm the user is attempting to build. Clicking on the "Application-OBJ" icon corresponds to building the application update algorithm and clicking on a "SUBSYSTEM-OBJ" icon corresponds to building a subsystem update algorithm. From here the user can choose to either automatically or manually build the update algorithm. To build the update algorithm automatically, the user must click on the "Update Algorithm" window surface and choose "Automatically Build Update Algorithm" from the menu displayed. AVSI III uses the import/export connections specified when making the import/export connection to infer an update algorithm. AVSI III then displays the inferred update algorithm and the user can modify or accept it. Manually building the update algorithm is identical to the previous version of AVSI (3). Using AVSI III to create a six object update algorithm, the application specialist performs one user action, thus reducing the application specialist's workload from 12 actions to one.

3.2.5 Checking semantics, executing the application, and saving the application.

3.2.5.1 Description of Current Semantic Checking & Application Execution

Process. Once the application has been built, the application specialist needs to check the semantics of the application. This is accomplished by clicking on the “Check Semantics” button of the control-panel. The results from the semantic check routine are displayed in the EMACS window and the AVSI control panel.

After the application has been semantically checked, the application specialist can execute it. This is accomplished by clicking on the “Execute Application” button on the control-panel. If the application has a non-event-driven-sequential, the execution simply proceeds and the results are displayed in the EMACS window. If the application has an event-driven execution mode, a text window displaying the current contents of the event queue is presented. The window, shown in Figure 3.6, also contains four control selections that allow the application specialist to add events to the queue through a set of input menus, modify events on the queue, or delete events from the queue. When all changes have been made, “Begin Execution” can be selected from the window. Execution then begins and the results are displayed in both the EMACS window and the AVSI control panel.

3.2.5.2 Analysis of Semantic Checking & Application Execution Process.

Until now, all buttons with the exception of the Save Application on the control-panel have been duplicated via modifications to the “Icon Operations” menu. To further increase usability, reducing the distance when moving the mouse pointer can be accomplished by

Event Queue for NANDGATE-TEST					
Event Type	For Primitive	Thru-Subsystems	Priority	Time	Comment
START-EVENT-OBJ	EVENT-HANDLER	(APP-EXEC)	100	0	
SET-STATE-EVENT-OBJ	SW-1	(SUB-1)	5	0	
SET-STATE-EVENT-OBJ	SW-2	(SUB-1)	5	50	
SET-STATE-EVENT-OBJ	SW-1	(SUB-1)	5	100	
SET-STATE-EVENT-OBJ	SW-2	(SUB-1)	5	150	
SET-STATE-EVENT-OBJ	SW-1	(SUB-1)	5	200	
SET-STATE-EVENT-OBJ	SW-2	(SUB-1)	5	200	
STOP-EVENT-OBJ	EVENT-HANDLER	(APP-EXEC)	100	300	
ADD an Event to the Queue MODIFY an Event on the Queue DELETE an Event from the Queue BEGIN EXECUTION - editing complete Display Old Event List (After Execution)					

Figure 3.6 Application Executive Event Queue

modifying the "Icon Operations" menu to also include the "Check Semantics", "Execute Application", and "Save Application" commands.

3.2.5.3 Improvements to Semantic Checking & Application Execution Process.

AVSI III provides the application specialist with an alternate way to check semantics, execute the application, and save the application. Clicking on any icon in the "System Composition Window" brings up the "Icon Operation" window. A selection can be made to execute any of the tasks mentioned above. This arrangement ensures complete redundancy for all operations relating to defining an application.

3.3 Application Executive Visual Capabilities

This section describes the current and proposed capabilities of the application executive. Here the enhancements of the application executive metaphor set and the application executive event queue is explained.

3.3.1 Application Executive Metaphor Set.

3.3.1.1 Description of the Current Application Executive Metaphor Set.

With AVSI II, creating a new application or loading a saved application that is either event-driven-sequential or time-driven-sequential execution mode, the application executive is automatically inserted into the application definition. This insertion is done without any feedback given to the application specialist.

3.3.1.2 Analysis of the Current Application Executive Metaphor Set. To

exhibit the correct behavior, visual representation of the application executive primitives

is essential. When an operation is done, it is always important that some sort of visual feedback is given to the application specialist to reinforce that the operation was completed. In this case, displaying icons that represent the components (Event Handler, Connection Manager, and Global Timer) of the Application Executive reinforces the fact that the Application Executive was inserted into the application definition.

3.3.1.3 Improvements to the Application Executive Metaphor Set. Visual feedback was added to AVSI III by creating icons for the “Application Executive” domain. The connection manager, clock, and event-handler icons, as shown in Figure 3.7, were developed using Cossentine’s Developing a Metaphor Set (3). This modification gave AVSI III the ability to visually represent the appropriate behavior specified by the application specialist.

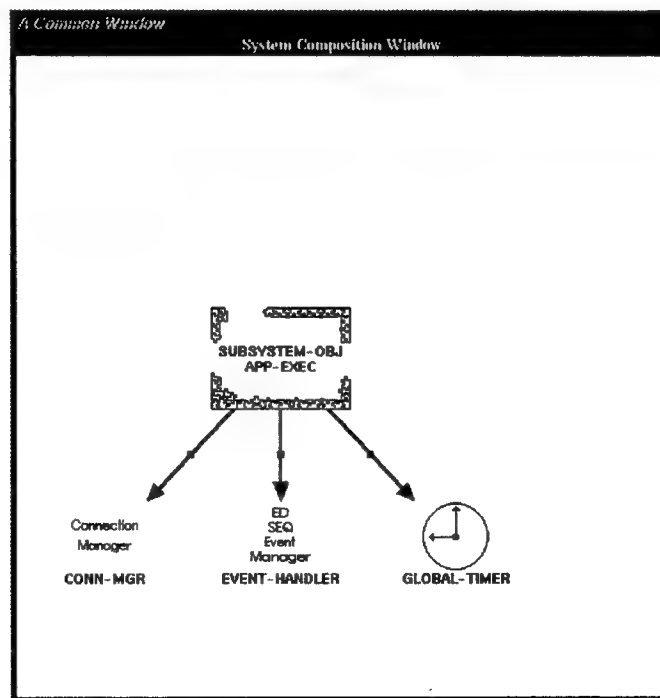


Figure 3.7 Application Executive Metaphor Set

3.3.2 Application Executive Event Queue.

3.3.2.1 Description of the Current Application Executive Event Queue Process. With the application built and semantically checked, executing an application using the event-driven-sequential or time-driven-sequential execution mode displays a text window displaying the current contents of the event queue. The window, shown in Figure 3.6, allows the application specialist to add, delete, or modify an event. Once all changes have been made, selecting "BEGIN EXECUTION" starts the execution of the application. Execution is simulated by displaying the contents of the event queue both before and after execution, thereby giving the application specialist a minimal snapshot of the behavior.

3.3.2.2 Analysis of the Application Executive Event Queue. To furnish a better behavioral picture of an application executing in the event-driven-sequential or time-driven-sequential mode, each change in the event queue is made visible in order that the application specialist may understand the true behavior of the application.

3.3.2.3 Improvements to the Application Event Queue. In AVSI III, whenever an event is added, modified, or deleted the change is made visible to the application specialist. With this revision, a more precise picture of the application behavior is exhibited.

3.4 Summary

AVSI II, the Architect Visual System Interface created by Weide (23) and improved upon by Cossentine (3), was essential in developing Architect into a favorable tool for users. While Weide based his research and implementation on functionality, and Cossentine based his research on the appearance and usability of the interface, this research effort concentrated on reducing the level of effort. AVSI III allows the application specialist to undertake the task of representing and executing an application with greatly reduced effort and in a more logical manner than he could with AVSI II.

The next chapter provides information relevant to the design decisions and tradeoffs experienced in developing AVSI III, as well as the rationale behind those decisions.

IV. Design and Implementation of AVSI III

4.1 Introduction

This chapter discusses the specifics of each aspect of AVSI III's design approach, design tradeoffs, and implementation details. It presents an analysis of the design decisions, and an examination of the implementation goals and objectives.

4.2 Design Approach

The basic concepts and procedures used in improving AVSI III were developed by applying common methods of software engineering. First, a basic understanding of AVSI II and the underlying structure of Architect was obtained. The design of AVSI II allowed the application specialist to compose applications by following a sequence of user actions. The main thrust of the design approach was based on reducing the level of effort required to accomplish each task. This can be broken down into several aspects of the user interface design. An analysis of the user tasks involved in building an application was used to determine the high-frequency tasks. High-frequency tasks are defined as tasks that are used repetitively. Once these high-frequency tasks were identified, a plan was developed based on the task.

The secondary thrust of the design approach was based on improving the application executive execution phase. Emphasis was placed on giving the application specialist an improved behavioral picture of the application. With the short time given to accomplish such a task, concentration was placed on improving the visual representation of the event queue.

4.2.1 Level of Effort. The design of AVSI II was analyzed to determine the high-frequency tasks. This was simplified by further decomposition in functional areas.

4.2.1.1 User Actions. To determine how to reduce the number of user actions required to accomplish a task, an investigation was done to determine which user actions were necessary and what improvements could be made. The identified user actions were classified into several functional areas. Each user action had its own set of modifications needed in order to improve the usability of AVSI. These functional areas are:

1. Create/Edit Application
2. Build Application Update Algorithm
3. Create Subsystems/Build Imports-Exports
4. Build Subsystem Update Algorithm
5. Execute the Application

4.3 Implementation

4.3.1 Create/Edit Application. This functional area was enhanced by analyzing the sequence of user tasks required and implementing several modifications to increase usability. To create and edit an application these user actions are necessary:

1. Click on the control-panel button labeled **Create New Application**.
2. Select Domain.
3. Enter Application Name.

4. Choose Execution Mode.
5. Click on Edit Application button.
6. Choose Application to edit.
7. Select Edit Application Components.
8. Invoke Window Operations Menu.
9. Choose Create New Subsystem.
10. Enter Subsystem Name.
11. Position Subsystem.
12. Invoke Icon Operations Menu.
13. Choose Link to Source.
14. Click on Source Icon.

The high-frequency user actions found for this task were selecting the execution mode and navigating the control-panel menu. The enhancements to reduce user actions in this task were:

1. Automatic Selection of Execution Mode
2. Reorder Window Presentation

4.3.1.1 Automatic Selection of Execution Mode. A simple approach was taken to enhance usability for this task. The execution mode selection was made automatic, instead of prompting the user to select from a list of execution modes (see section 3.1). The implementation of this enhancement was accomplished by replacing the single-menu

function with a newly designed `get-execution-mode` function. The code representing the pop-up window prompting for the execution mode was:

```
let(exec-mode :symbol = single-menu( "Choose Execution Mode: ", exe-modes))  
  app-mode(desc-obj) <- exec-mode;
```

and was replaced with:

```
app-mode(desc-obj) <- get-execution-mode(*CURRENT-DOMAIN*);
```

The `get-execution-mode` function was defined as follows:

```
function get-execution-mode(this-domain: symbol): symbol =  
  
  let (exec-mode: symbol = undefined)  
  
    this-domain = 'CIRCUITS  
      --> ( exec-mode <- 'NON-EVENT-DRIVEN-SEQUENTIAL );  
    this-domain = 'ED-CIRCUITS  
      --> ( exec-mode <- 'EVENT-DRIVEN-SEQUENTIAL );  
    this-domain = 'DSP  
      --> ( exec-mode <- 'NON-EVENT-DRIVEN-SEQUENTIAL );  
    this-domain = 'CRUISE-MISSILE  
      --> ( exec-mode <- 'TIME-DRIVEN-SEQUENTIAL );  
  
  exec-mode
```

This code is fairly easy to understand and maintain.

4.3.1.2 Reorder Window Presentation. The idea here was to analyze the sequence of windows and the number of user actions required to navigate the required windows to achieve the necessary task. Creating a new application always begins with creating a controlling subsystem. The goals of this section are:

1. Establish the System Composition Window as the Primary window.

2. Along with the System Composition Window expose the Technology Base Window and enable the user to create instances of primitive objects.
3. Duplicate application functions in easier to access menus. Easier in the sense that less user actions are required. This can be accomplished by making some selections context-sensitive.
4. Reduce the distance that the user has to move the mouse to access frequently used menus and commands.
5. Give the user the option between two methods to navigate the menus.

The reordering of windows was based on two main portions of the AVSI II code. The fact that creating an application always began by defining a controlling subsystem was the first portion. Loading a saved application, was frequently followed by modifying the application. In both cases, presenting the `edit-objects` window automatically reduces the number of user actions the application specialist needed. In creating a new application, adding a call to the `edit-appl-components` function in `create-new-application` function automatically presents the "Application-Obj" window. The same goes for loading a saved application, adding a call to the `edit-appl-components` function in `load-saved-application` function automatically presents the "Application-Obj" window.

Another user action reduction strategy used was to give the application specialist the opportunity to add and link primitives while also creating and linking subsystems in the "Application-Obj" window. This meant displaying the "Technology Base Window" while also displaying the "Application-Obj" window. This enhancement just required modifying the `edit-appl-components` function to include calls to:

```
expose-window(*TECH-BASE-WINDOW*);
```

```
display-primitives();
```

The mouse handler function for the technology base window already allows the user to create instances of primitive objects chosen from the technology window and place them into the "Application-Obj" window, so the mouse handler did not need any modifications. With both the "Application-Obj" and the "Technology Base Window" windows exposed, the application specialist is allowed to instantiate, delete, and link primitive objects and subsystems in one convenient location without having to traverse more windows.

With the above changes, the original function of the "Application-Obj" window was changed moderately so a change to the window title from "Application-Obj" to "System Composition Window" was appropriate. This change was done in the `edit-objects` function in the refine file `edit-ss.re`.

```
window-title(dw) <- concat(symbol-to-string(name(instance-of(ss))), ": ",
                             symbol-to-string(name(ss)));
```

was replaced with

```
window-title(dw) <- ("System Composition Window");
```

To further decrease user actions for the application specialist, a design decision to force the user to **default** windows provided a position from which all user actions could start from. These **default** windows are the "System Composition" and "Technology Base" windows. For example, whenever the application specialist deactivates any window other than the **default** windows, AVSI III deactivates all active windows and redisplay the "System Composition" and "Technology Base" windows. The application specialist is always brought back to the same starting point.

Using AVSI III to create and edit an application the following 10 vice 14, user actions are necessary:

1. Click on the control-panel button labeled **Create New Application**.
2. Select Domain (Execution mode is inferred).
3. Enter Application Name.

(a) System Composition Window & Technology Base Window automatically displayed.

4. Invoke Window Operations Menu.
5. Choose **Create New Subsystem**.
6. Enter Subsystem Name.
7. Position Subsystem.
8. Invoke Icon Operations Menu.
9. Choose **Link to Source**.
10. Click on Source Icon.

4.3.2 Create Subsystems/Build Imports-Exports. To create additional subsystem(s) and/or primitive(s) and to connect the import and export connections these user actions were necessary in AVSI II:

1. Click on the control-panel button labeled **Edit Subsystem**.
2. Select the subsystem to edit.

3. Click on Objects icon on OCU model.
4. Create New Subsystems as described in Section 4.3.1.
5. Click on primitive object in Technology Base Window
6. Place in Subsystem-Obj window
7. Enter a name for instance of primitive object.
8. Repeat for other primitive objects.
9. Click on instance of primitive object (invokes Icon Operations Menu).
10. Select Link to Source.
11. Click on Subsystem source.
12. Deactivate window.
13. Click on Import or Export area icon on OCU model.
14. Choose on Make Connections.
15. Click on export icon.
16. Click on import icon.

The high-frequency user actions found for this task were navigating the control-panel menu to make the import and export connections of the corresponding primitives and subsystems. The enhancements to reduce user actions in this task were:

1. Icon Operations Menu
2. Imports/Exports Diagram Window

4.3.2.1 Icon Operations Menu. The icon operation menu modification was fundamental in reducing the number of user actions necessary for the application specialist to define an application. Decreasing the distance the application specialist had to move the mouse pointer in order to select an operation was the first consideration. Decreasing the number of menu levels was also another viable design consideration in reducing the user actions required. Adding menu items to the icon menu was a simple task in that adding the following lines of code was sufficient.

```
<"Make Connections", (lambda(i, w) make-connections(current-application))>,  
<"Edit Application Update", (lambda(i, w) edit-appl-update(tree-node-for-icon(i)))>,  
<"Check Semantics", (lambda(i, w) semantics-check-application())>,  
<"Execute Application", (lambda(i, w) execute-the-application())>,  
<"Save Application", (lambda(i, w) save-appl())>]
```

The mouse handler allows a vehicle for object class identification. That vehicle can be used to define context-sensitive function calls. For example, if the application specialist clicks on a subsystem icon, subsequent calls to `make-connections` or `edit-appl-update` are made with that subsystem as its parameter.

4.3.2.2 Imports/Exports Diagram Window. The modifications to the imports/exports connection process was accomplished by providing the application specialist with an alternative way to navigate thru the hierarchical control-panel menus. This alternative way to navigate thru the menus involves using the Icon Operations menu described in section 4.3.2.1. The "Make Connections" selection on the Icon Operations menu, provides the application specialist with a direct link to the Imports/Exports diagram window. Clicking on a subsystem icon displays an Imports/Exports diagram window for that subsystem. All subsystems and/or primitives can then be defined and linked to their

controlling subsystem. All imports can also be connected to their corresponding exports and vice versa. Using AVSI III to create additional subsystem(s) and/or primitive(s) and to connect the import and export connections the following 8 vice 16 user actions were necessary:

1. Click on the controlling subsystem icon.
2. Select **Make Connections**.
 - (a) The Imports/Exports Diagram window for the controlling subsystem is displayed.
3. Create New subsystems/primitives as described in section 4.3.1.
 - (a) Each subsystem and primitive is linked to the controlling subsystem.
4. Deactivate window.
5. Click on Import or Export area icon on OCU model.
6. Choose on **Make Connections**.
7. Click on export icon.
8. Click on import icon.

4.3.3 Building Application/Subsystem Update Algorithms. When using AVSI II, creating an application with the non-event-driven-sequential execution mode required the application specialist to define the update algorithm. The update algorithm specifies the sequence in which the primitive objects or subsystems in the application update their state information so that the data is propagated through the application to the output in the

appropriate order. To decrease the load on the application specialist in AVSI III, when the application specialist chooses to "Edit Update Algorithm", the user is given an option to have the update algorithm built automatically.

4.3.3.1 Specifying the Update Algorithm. With AVSI II, to edit the application update algorithm these user actions were necessary:

1. Click on the control-panel button labeled **Edit Application**.
2. Choose Application to edit.
3. Edit Application Update.
4. Click on Subsystem Icon.
5. Place in Application Update Sequence.
6. Invoke Window Operations Menu.
7. Select Deactivate.

With AVSI II, to edit the subsystem update algorithm these user actions were necessary:

1. Click on the control-panel button labeled **Edit Subsystem**.
2. Choose Subsystem to edit.
3. Click on Controller Icon.
4. Click on Subsystem/Primitive Icon.
5. Place in Algorithm Update Sequence.

6. Invoke Window Operations Menu.
7. Select Deactivate.

The enhancements involved in reducing user actions in this task were:

1. Reorder Window Presentation
2. Modify Icon Operations Menu
3. Infer update algorithm
4. Window Operations Menu

4.3.3.2 Reordering of Window Presentation. Because AVSI III defaults to the "System Composition" and "Technology Base" windows, the application specialist does not have to navigate further before he can invoke the "Icon Operations" window. Clicking on an application icon or a subsystem icon causes AVSI III to display the Icon Operations menu. Selecting "Edit Update Algorithm" first executes `edit-appl-update`, which causes an evaluation of the `menu-y-or-n?("Infer Update Algorithm?")` condition. `Menu-y-or-n?` is an INTERVISTATM function that provides a special type of single-choice menu that is appropriate when you want to ask a question that can be answered with a **yes** or a **no** (17). Answering **yes** calls `infer-update-algorithm(app)` which is the newly defined function designed to infer the update algorithm.

4.3.3.3 Modify Icon Operations Menu. These enhancements have already been described in section 4.3.2.1.

4.3.3.4 *Infer Update Algorithm.* The derivation of the update algorithm attains a tremendous decrease in user actions required by the application specialist. The next section outlines implementation code conceived for this purpose.

```
function infer-update-algorithm(app: application-obj) =

    format(factive, "%Entering the Infer-Update-Algorithm function in edit-applic.re~%");

    format(debug-on, "~%APP is: ~A~%", app);

    let(new-statement: statement-obj = undefined,
        i-obj: import-obj = undefined,
        o-obj: export-obj = undefined,
        input-set: set(symbol) = {},
        output-set: set(symbol) = {},
        input-output-set: set(symbol) = {},
        sequence-set : set(symbol) = {},
        sequence: seq(statement-obj) = [],
        top-ss: subsystem-obj = arb({x | (x: subsystem-obj)
                                     (subsystem-obj(x)) &
                                     (~empty(Import-Area(x)))})

    ) % let

    %-----
    % If requesting to build the application update algorithm,
    % typically there is only one top level subsystem so, the
    % the algorithm is trivial. Note: If there is more than
    % one top level subsystem then this code relies on the
    % user to specify the sequence of execution.
    %-----

    application-obj(app)

    --> ((enumerate o over application-components(app) do
        new-statement <- make-object('update-call-obj);
        operand(new-statement) <- o;
        sequence <- prepend(sequence, new-statement)
    ); % End Enumerate

    size(application-components(app)) = 1
```

```

--> application-update(app) <- sequence
    ); % End Transform

%-----
% Requesting to build a subsystem update algorithm.
%-----
subsystem-obj(app)
--> ((enumerate o over controllees(app) do

    (enumerate i-obj over Import-Area(top-ss) do
        (consumer(i-obj) = o) or
        (subsystem-obj(find-object('subsystem-obj, o))) --> o in input-set);

    (enumerate o-obj over Export-Area(top-ss) do
        (producer(o-obj) = o) or
        (subsystem-obj(find-object('subsystem-obj, o))) --> o in output-set));

input-output-set <- input-set intersect output-set;
input-set        <- setdiff(input-set, input-output-set);
output-set       <- setdiff(output-set, input-output-set);

while ~empty(input-output-set) do
    (enumerate i-obj over input-output-set do
        let(set-import-obj : set(import-obj) = {x |
            (x: import-obj)
            (import-obj(x))      &
            (x in Import-Area(top-ss)) &
            (consumer(x) = i-obj)},
            no-exports-in-sub : boolean = true)

        (enumerate o-obj over set-import-obj do
            Source-Subsystem(arb(sources(o-obj))) = name(app) &
            (Source-Object(arb(sources(o-obj))) ~in sequence-set)
            --> (no-exports-in-sub <- false));

no-exports-in-sub
--> (new-statement <- make-object('update-call-obj);

```

```

        operand(new-statement) <- i-obj;
        sequence <- append(sequence, new-statement);
        t --> i-obj ~in input-output-set;
        t --> i-obj in sequence-set)

);

(enumerate o-obj over output-set do
  new-statement <- make-object('update-call-obj);
  operand(new-statement) <- o-obj;
  sequence <- prepend(sequence, new-statement));

%-----
% Check where in the update algorithm to place the objects
% that have both imports & exports
%-----

(enumerate x over input-output-set do
  let(import-set : set(import-obj) = {},
      update-flag : boolean = undefined)

    % -----
    % Build a set "import-set" of import-objs that connect with
    % primitive "x"
    %-----

    (enumerate y over import-area(top-ss) do
      (consumer(y) = x) --> y in import-set);

    update-flag <- true;

    %-----
    % Build a set "sequence-set" of symbols of objects that are
    % already in the update algorithm
    %-----

    (enumerate y over sequence do
      t --> operand(y) in sequence-set);

```

```

%-----
% Check each import-obj to see if the source of that import is
% already in the update algorithm "sequence"
%-----
(enumerate y over import-set do
  (Source-Object(arb(Sources(y))) ~in sequence-set)
  --> (update-flag <- false));

update-flag
--> (new-statement <- make-object('update-call-obj);
    operand(new-statement) <- x;
    t --> sequence <- append(sequence, new-statement);
    t --> x ~in input-output-set));

(enumerate i-obj over input-set do
  new-statement <- make-object('update-call-obj);
  operand(new-statement) <- i-obj;
  sequence <- append(sequence, new-statement));

update(app) <- sequence;

format(debug-on, "%Exiting the Infer-Update-Algorithm function in edit-applic.re%")

```

4.3.3.5 *Window Operations Menu.* Cossentine's modification that involved closing all the subwindows simultaneously was extended. The application specialist using AVSI II could be confused by the fact that to deactivate all the windows simultaneously, the procedure to invoke the "Window Operations" menu was inconsistent. To invoke the "Window Operations" menu from the update algorithm window required clicking on the window *border*. To invoke the "Window Operations" menu from the Imports/Exports window and the Composition window required a click on the blue window *background*.

AVSI III clears this inconsistency by providing uniformity with regards to the "Window Operations" menu. To invoke the "Window Operations" menu from the Update Algorithm window, Imports/Exports window, or the System Composition window now requires a click on the window *background*. The following REFINETM code was inserted to implement this modification.

```
diagram-surface(obj)
    --> Abbrev-Update-Win-Mouse-Handler(event, obj, pos, dw);
```

With AVSI III, to edit the application/subsystem update algorithm the following 5 vice 7 user actions are necessary:

1. Click on the Application/Subsystem icon.
2. Choose YES to automatically build update algorithm. OR
3. Choose NO to manually build update algorithm.
 - (a) Click on Subsystem/Primitive Icon.
 - (b) Place in Algorithm Update Sequence.
 - (c) Repeat steps 3a and 3b for each subsystem/primitive in the application/subsystem.
4. Invoke Window Operations Menu.
5. Select Deactivate.

4.3.4 Application Executive Visualization. AVSI II gave the application specialist a before and after view of the behavior of an event-driven-sequential or time-driven-sequential execution mode application. Visual feedback was limited to displaying the event

queue before execution and displaying the event queue after execution completed. The enhancement implemented in AVSI III gives the user a fuller picture of the behavior that the defined application exhibits from the instant that execution begins. The enhancements involved in improving the event queue were made to the `app-exe.re` source file. A very simplistic approach was used in this revision of the event queue. The `display-queue` function was placed in the portions of the code that affects the state of the event queue. It was determined that whenever an event was added, modified, or deleted from the event queue, the event queue should be redisplayed.

4.4 Conclusion

This chapter described the motivation behind the design of AVSI III and the details of its implementation. The design goals of AVSI III were to reduce the level of effort and increase usability for an application specialist building an application with Architect while preserving the AVSI functionality. The next chapter describes the evaluation criteria, the evaluation results, and the analysis of the evaluation results.

V. Testing and Validation of AVSI III

5.1 Introduction

Validation of AVSI III consisted of compatibility and usability testing. The compatibility and functionality of AVSI III were verified by using the interface to compose and execute the applications that were constructed prior to AVSI III development. The usability of the user interface was tested with an objective analysis of the reduction in workload. This chapter reviews the effort reduction, usability, and functionality testing of AVSI III, the results of the testing, and an analysis of the test results. Additionally, an analysis of the strengths and weaknesses of AVSI III is provided.

5.2 Validation Domains

AVSI III was tested using previous applications developed in the following domains:

1. Digital Logic-Circuits
2. Digital Signal Processing (DSP)
3. Event-Driven Circuits
4. Cruise Missile

The user interface was informally tested throughout its development by other KBSE researchers. Because the domains used for testing were previously validated (1, 16, 23, 3, 4, 21, 22, 24), it was not necessary to test the operational capabilities of the domain primitives. Thus, the test suites were designed to verify the user interface capability.

5.3 Testing

5.3.1 Compatibility Testing. The purpose of this portion of the testing was to prove that AVSI III is completely backwards compatible with *SAVED* applications that were previously defined using AVSI II, AVSI, and the Architect command-line interface.

All previously defined applications were loaded and viewed in the system-composition window, the update-algorithm-window, and the imports-exports-window to verify the appropriate visual information was displayed and correct behavior of the application was represented. The applications were then executed and validated against the behavior expected. All the applications were then saved and the resulting save files were compared to the previously defined files to verify data integrity.

The results of the compatibility testing showed that, with the exception of applications in the *Cruise-Missile* domain, all previously defined applications could be composed using the new procedures of AVSI III. The irregularities only occurred in the *Cruise-Missile* domain applications when trying to execute the applications. This anomaly was attributed to an unresolved discrepancy from previous research.

5.3.2 Measuring Usability Improvement. Usability improvement was based on counting the number of distinct user actions required to build an application. These distinct actions include, but are not limited to mouse clicks and keyboard inputs. A mouse click is easily distinguishable from another mouse click, but a distinct keyboard input is defined as a finite number of keystrokes that is necessary to answer a needed requirement from Architect. Examples of a distinct keyboard input are:

1. Name of application
2. Name of subsystem(s)
3. Name of primitive(s)
4. Value for SW-OBJ-POSITION

Several previously defined applications of differing complexity and execution modes were used for comparison. Each previously defined application was recreated using both AVSI II and AVSI III to determine the number of user actions required to build that application. Those results were then analyzed to determine improvements to usability.

5.3.3 Digital Logic Circuits Domain. The Digital logic circuit domain is executed in the non-event-driven-sequential execution mode.

5.3.3.1 Exclusive-Or. The first test application developed was a four Nand-gate implementation of an Exclusive-OR gate. The component layout and truth table for the circuit are shown in Figure 5.1. The total number of primitives add up to seven. Table 5.1 shows the number of user actions required to build the application. The results indicate a 48.8% decrease in level of effort from AVSI II to AVSI III.

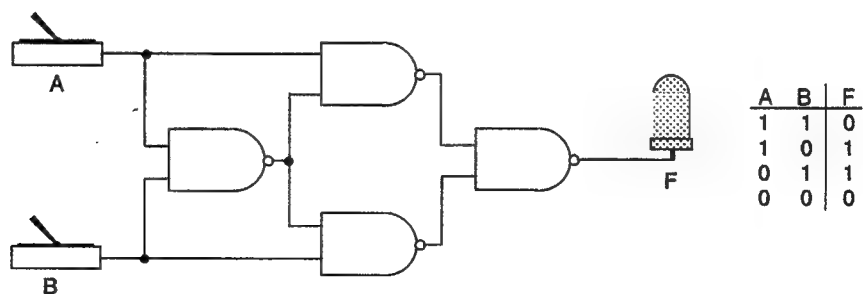


Figure 5.1 Exclusive-OR Circuit and Truth Table

Table 5.1 Exclusive-OR Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Application Update	7	5
Create Subsystems/Build Imports-Exports	79	43
Subsystem Updates	22	5
Execute	1	1
TOTAL	125	64

5.3.3.2 Half Subtractor. The second test application was a seven gate implementation of a Half Subtractor. The component layout and truth-table for the circuit are shown in Figure 5.2. The total number of primitives add up to twelve. Table 5.2 shows the number of user actions required to build the application. The results indicate a 44.9% decrease in level of effort from AVSI II to AVSI III.

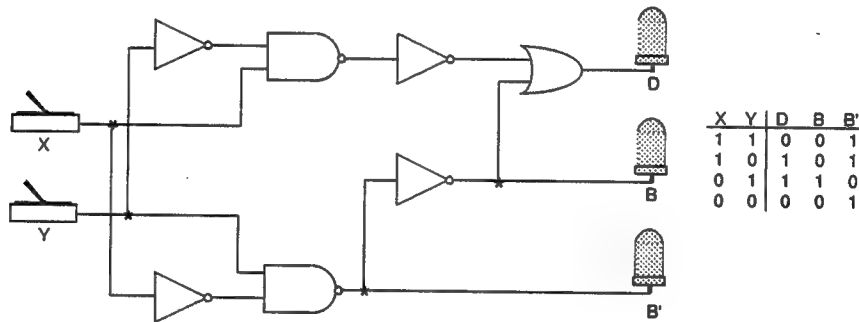


Figure 5.2 Half Subtractor Circuit and Truth Table

5.3.3.3 Binary Array Multiplier. The third test application was a four gate, two half-adder implementation of a Binary Array Multiplier. The component layout and truth-table for the circuit are shown in Figure 5.3. The total number of primitives add up to fourteen. Table 5.3 shows the number of user actions required to build the application. The results indicate a 38.3% decrease in level of effort from AVSI II to AVSI III.

Table 5.2 Half Subtractor Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Application Update	7	5
Create Subsystems/Build Imports-Exports	102	66
Subsystem Updates	32	5
Execute	1	1
TOTAL	158	87

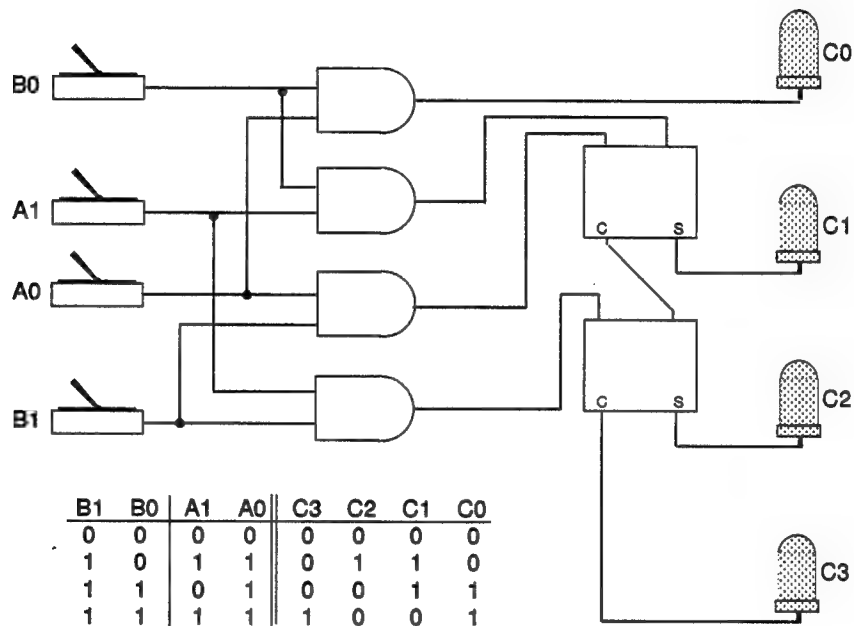


Figure 5.3 Binary Array Multiplier Circuit and Truth Table

Table 5.3 Binary Array Multiplier Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Application Update	7	5
Create Subsystems/Build Imports-Exports	107	82
Subsystem Updates	36	5
Execute	1	1
TOTAL	167	103

5.3.3.4 *BCD to Excess-3 Decoder.* The fourth test application was an eleven gate implementation of a BCD-to-Excess-3 Decoder. The component layout and truth table for the circuit are shown in Figure 5.4. The total number of primitives add up to nineteen. Table 5.4 shows the number of user actions required to build the application. The results indicate a 29.3% decrease in level of effort from AVSI II to AVSI III.

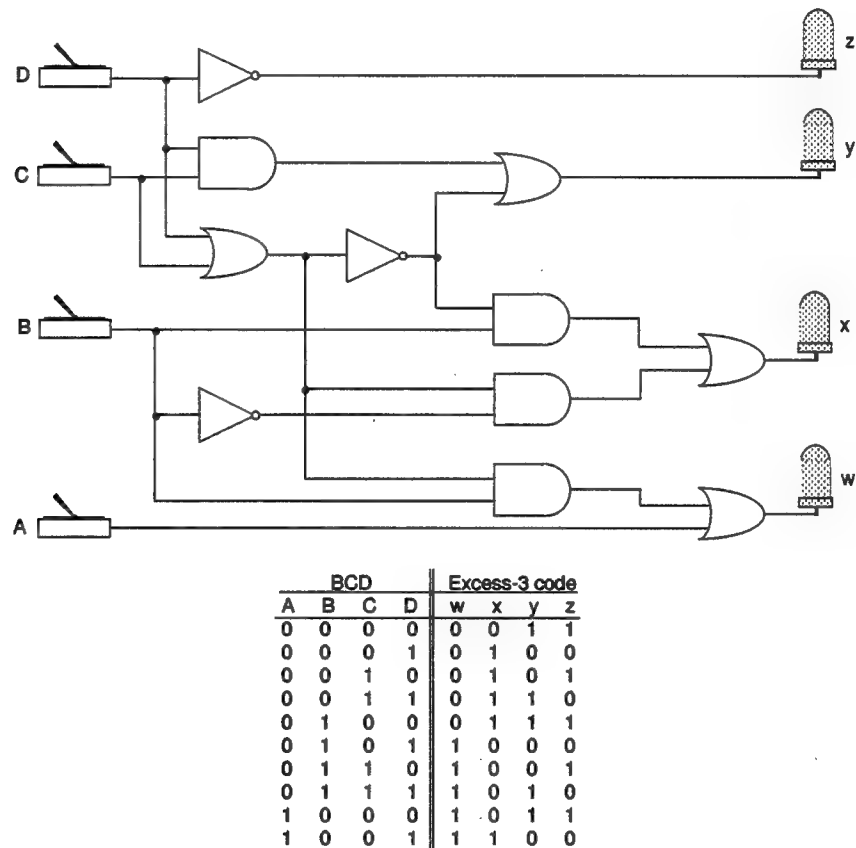


Figure 5.4 BCD to Excess-3 Decoder Circuit and Truth Table

5.3.4 *Digital Signal Processing Domain.* The Digital signal processing domain is also executed in the non-event-driven sequential execution mode.

Table 5.4 BCD to Excess-3 Decoder Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Application Update	7	5
Create Subsystems/Build Imports-Exports	116	107
Subsystem Updates	41	5
Execute	1	1
TOTAL	181	128

5.3.4.1 Signal-With-Noise. The fifth test application was composed of a sinusoid signal, a noise signal, a signal adder, and a one-input graph display implementation of a Signal-With-Noise. The component layout for the circuit is shown in Figure 5.5. The total number of primitives add up to four. Table 5.5 shows the number of user actions required to build the application. The results indicate a 29.5% decrease in level of effort from AVSI II to AVSI III.

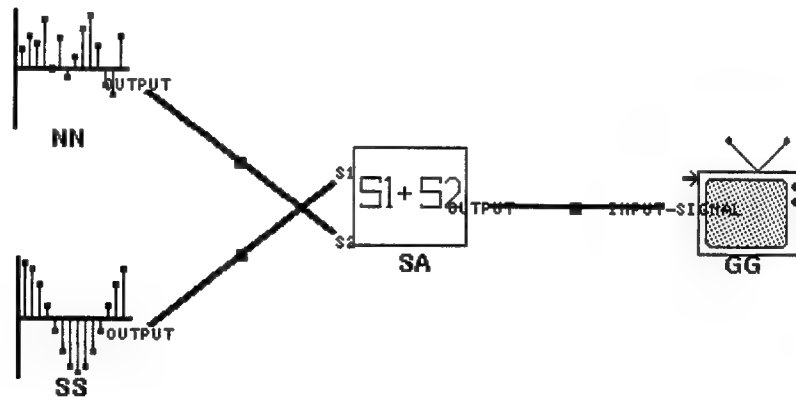


Figure 5.5 Signal-With-Noise DSP

5.3.4.2 Window-Demo. The sixth application, called Window-Demo, was composed of a sinusoid signal, a window signal, two Discrete Fourier Transform primitives,

Table 5.5 Signal-With-Noise Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Application Update	7	5
Create Subsystems/Build Imports-Exports	26	22
Subsystem Updates	11	5
Execute	1	1
TOTAL	61	43

two complex-to-real converters, and a two-input graph display implementation. The component layout for the circuit is shown in Figure 5.6. The total number of primitives add up to seven. Table 5.6 shows the number of user actions required to build the application. The results indicate a 32.6% decrease in level of effort from AVSI II to AVSI III.

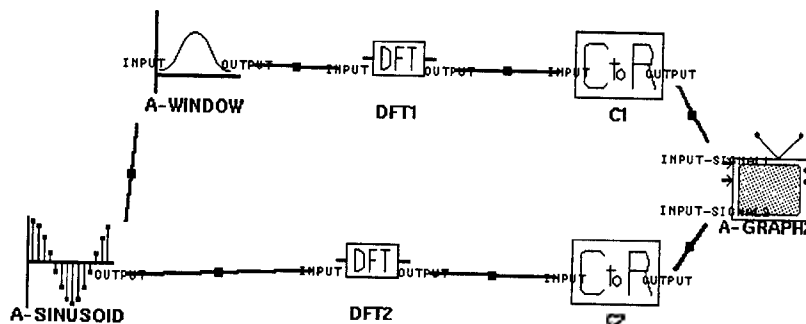


Figure 5.6 Window Demo

Table 5.6 Window-Demo Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Application Update	7	5
Create Subsystems/Build Imports-Exports	48	39
Subsystem Updates	17	5
Execute	1	1
TOTAL	89	60

5.3.4.3 *Four-Sum-Moving-Average.* The seventh application was a DSP application called a moving average, shown in Figure 5.7. This particular application is a four-sum moving average: each sample of the output is the average of the current output along with the three previous samples. The multiplier primitive (looks like a triangle) has a value of 0.25 for its multiply value. The input signal loaded from a file was generated by a previous application that simply added a sinusoid with some noise. The total number of primitives add up to eleven. Table 5.7 shows the number of user actions required to build the application. The results indicate a 33.3% decrease in level of effort from AVSI II to AVSI III.

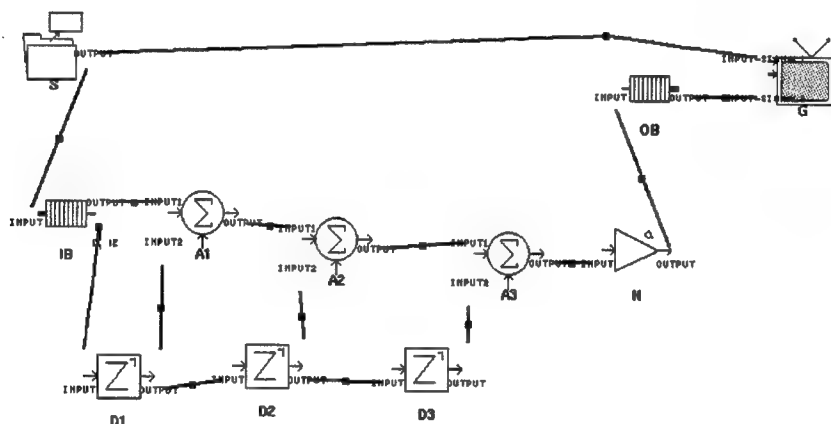


Figure 5.7 Four Sum Moving Average

Table 5.7 Four-Sum-Moving-Average Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Application Update	7	5
Create Subsystems/Build Imports-Exports	74	65
Subsystem Updates	31	5
Execute	1	1
TOTAL	129	86

5.3.5 *Event-Driven Circuits Domain.* The Event-Driven circuit domain is executed in the event-driven sequential execution mode.

5.3.5.1 *Half Adder.* The eighth test application contained one half-adder component. The component layout for the circuit is shown in Figure 5.8. The total number of primitives add up to five. Table 5.8 shows the number of user actions required to build the application. The results indicate a 27.7% decrease in level of effort from AVSI II to AVSI III.

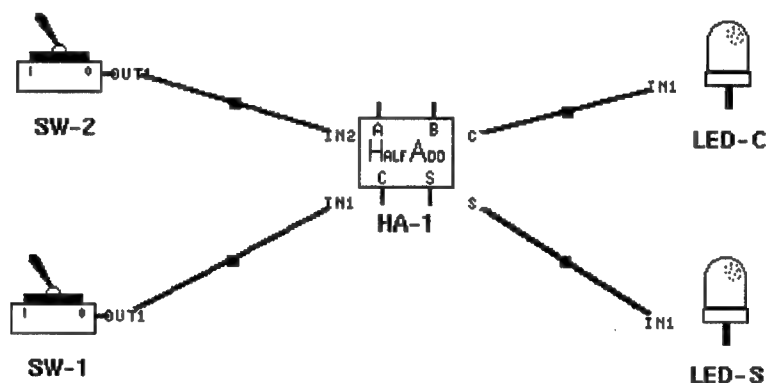


Figure 5.8 Half Adder Event-Driven Circuit

Table 5.8 Half Adder Event-Driven Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Create Subsystems/Build Imports-Exports	36	27
Execute	2	2
TOTAL	54	39

5.3.5.2 *Nand Gate.* The ninth test application contained one nand gate. The component layout for the circuit is shown in Figure 5.9. The total number of primitives

add up to four. Table 5.9 shows the number of user actions required to build the application.

The results indicate a 30.6% decrease in level of effort from AVSI II to AVSI III.

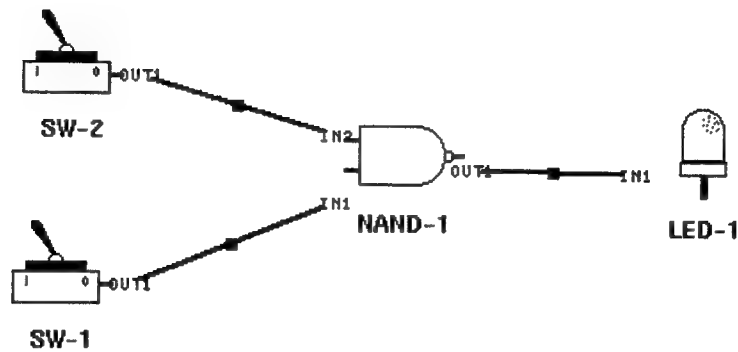


Figure 5.9 Nand Gate Event-Driven Circuit

Table 5.9 Nand Gate Event-Driven Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Create Subsystems/Build Imports-Exports	31	22
Execute	2	2
TOTAL	49	34

5.3.5.3 JK Flip Flop. The tenth test application contained one JK flip-flop circuit. The component layout for the circuit is shown in Figure 5.10. The total number of primitives add up to six. Table 5.10 shows the number of user actions required to build the application. The results indicate a 25.4% decrease in level of effort from AVSI II to AVSI III.

5.3.6 Cruise-Missile Domain. The Cruise-Missile domain is executed in the time-driven sequential execution mode.

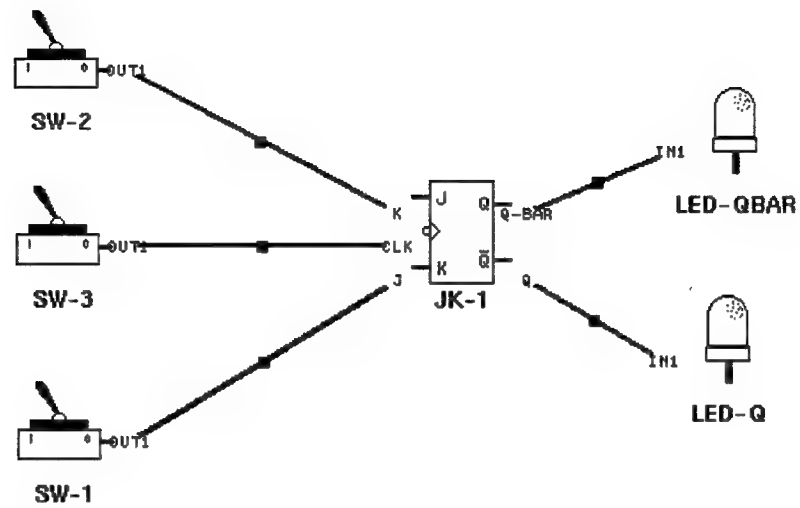


Figure 5.10 JK Flip Flop Event-Driven Circuit

Table 5.10 JK Flip Flop Event-Driven Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	16	10
Create Subsystems/Build Imports-Exports	41	32
Execute	2	2
TOTAL	59	44

5.3.6.1 Cruise Missile Flight. The eleventh and final application consists of four subsystems. The Warhead subsystem contains a warhead primitive. The Avionics subsystem contains a autopilot primitive, a guidance system, and a navigational system. The Airframe subsystem contains the Airframe primitive. The Propulsion subsystem contains an engine, a throttle, and a fuel tank primitive. The component layout for the cruise missile application is shown in Figure 5.11. This application contains four subsystems and eight primitives. Table 5.11 shows the number of user actions required to build the application. The results indicate a 33.5% decrease in level of effort from AVSI II to AVSI III.

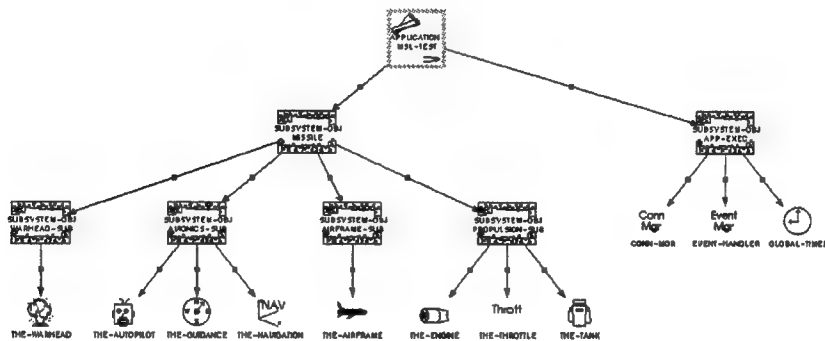


Figure 5.11 Cruise Missile

Table 5.11 Cruise Missile Flight Test Results

Sequence of operations	AVSI II	AVSI III
Create/Edit Application	37	35
Create Subsystems/Build Imports-Exports	152	90
Execute	2	2
TOTAL	191	127

5.4 Analysis

The results from the usability improvement test show a significant reduction in the level of effort an application specialist has to put forth when using AVSI III as opposed to AVSI II. The average reduction in workload was computed at 34.0% (see Table 5.12). The information gathered also showed that the more connections and primitives needed to build an application, the higher the savings would be. The most improvement was attributed to the automatic update algorithm functional area. The other functional area worth mentioning was the build imports/exports functional area.

Table 5.12 Usability Improvement

Execution Mode	Application	% Improvement
Non-event-driven sequential	Exclusive-OR	48.8
Non-event-driven sequential	Half-Subtractor	44.9
Non-event-driven sequential	Binary Array Multiplier	38.3
Non-event-driven sequential	BCD to Excess-3 Decoder	29.3
Non-event-driven sequential	Signal-With-Noise	29.5
Non-event-driven sequential	Window-Demo	32.6
Non-event-driven sequential	Four-Sum-Moving-Average	33.3
Event-Driven sequential	Half-Adder	27.7
Event-Driven sequential	Nand-Gate	30.6
Event-Driven sequential	JK Flip-Flop	25.4
Time-Driven sequential	Cruise Missile	33.5
AVERAGE		34.0

5.5 AVSI III's Shortcomings

The visual enhancements of AVSI III and the increased usability of the interface are significant accomplishments. However, there is always a new feature or a better layout for any display. The evolutionary development of Architect requires that the graphical

interface also evolve to incorporate new functionality. The following areas of AVSI III still require significant research:

1. Extend support for the Architect Application Executive. AVSI III provides only rudimentary support for the power of the Executive. The application specialist should be given the option to compose an Application Executive. Currently, Architect and AVSI III are designed to support the non-event-driven-sequential, event-driven-sequential, and time-driven-sequential execution modes with a variety of domains, but the application specialist can only compose applications that are of only one execution mode. To compose more diversified applications the ability to mix execution modes would extend the capabilities of Architect to realms currently unattainable.
2. Visualization of application execution. An object's state could be displayed using color, highlighting, sound, or movement. Color and highlight style are maintained as attributes of an icon, and an icon is maintained as a functional converse of an object. The state of an object is simply an attribute or collection of attributes of the object. Therefore, it is a straight-forward task to monitor the state of an object by checking its attributes, and set or modify the values of the corresponding icon's color or highlight style attributes. Movement of an icon can be simulated by repeatedly re-drawing the icon (and bitmap) at a sequence of positions. The speed at which the icon is redrawn and the distance between successive points will determine how smooth the movement appears.
3. Visual support for semantic analysis. AVSI currently uses Architect's semantic analysis reporting, which simply reports semantic errors in the Emacs window in textual

form. Though this is adequate for the current implementation, using visualization for semantic error reporting would provide a more user-friendly system. Such a use of visualization would bring up one or more of AVSI's editors in the "problem area". For example, if there was a problem with the definition of a subsystem's structure, the subsystem editor would be invoked, and would contain the subsystem in question, with the problem area highlighted. If there was a problem with an object's attributes, the object attribute editor would be invoked, with the attribute highlighted.

4. Incorporate more sophisticated graphical layout routines. The imports/exports window's visual layout algorithm is adequate, but incorporating a more complex icon layout routine would greatly increase usability.
5. The zooming capabilities of AVSI III only allows the user to set the size of the icons to one of two different sizes.

5.6 Conclusion

The Human Computer Interaction concepts applied to this research, and summarized in Chapter II, have shown that a more usable graphical user interface can be developed with a reduced level of effort to support a visual composition process. AVSI III was instrumental in enhancing the interface to Architect. The improved procedures for automatically building the update algorithms for non-event-driven-sequential execution applications and the expanded flexibility of the menu system reduce the user's workload and reduce the possibility of error from incorrect selection of execution modes. AVSI III proved to be fully backward compatible with previously designed applications, thus preserving the efforts of earlier research. This chapter reviewed the process by which AVSI III was tested

and validated. Some of AVSI III's shortcomings were mentioned. Those shortcomings, as well as some of the user feedback comments, are addressed in more detail in the next chapter.

VI. Conclusion and Recommendations

6.1 Overview

This chapter provides a summary of the accomplishments of this thesis effort. It also discusses the conclusions which can be drawn from this work and presents some recommendations for further research.

6.2 Results of This Research

AVSI III successfully achieved the original research goals. Design and development were conducted through rapid prototyping and incremental improvement. This philosophy proved to be highly successful. Each step in the research built upon earlier accomplishments and extended the capability of AVSI III. The challenge was to make AVSI III more usable. Accomplishment of the four research goals are summarized as:

1. Inferencing of the application/subsystem update algorithms by synthesizing a sequencing algorithm from visual representation of control data was achieved by using as a basis the import and export connections of the subsystem(s) and primitive(s) of the model.
2. The integration of displaying intermediate event queue states gives the application specialist a better behavioral understanding of the application being executed.
3. The modification of AVSI to support multiple domain application definitions (5).
4. Testing of the new interface changes and quantification of AVSI III's improvements was completed, and it validated the proposition that decreasing level of effort for the application specialist increases user interface usability.

6.3 Recommendations for Future Research

1. *Explore the use of sounds in the interface.* By making LISP calls to the host operating system, an interface developer can incorporate sounds into the interface design. Sounds are a common part of user interface design, and, although easy to misuse or abuse, sound can provide strong reinforcement or feedback to a user. Computers typically beep at a user if an error occurs, or chime upon successful completion of a task.
2. *Incorporate more sophisticated graphical layout routines* The imports/exports window's visual layout algorithm is adequate, but incorporating a more complex icon layout routine would greatly increase usability. This capability would give the application specialist a means to automatically reposition icons and their links with a minimum of link crossing. Example: Knowledge Based Software Assistant (KBSA) of Anderson Consulting(2).
3. *Provide AVSI with scalable zoom in and zoom out capabilities* The addition of scalable zooming capabilities would provide the application specialist with a better facility to review the application.
4. *Provide visual support for semantic checks* A more complete and user-friendly visual system would guide the user through finding mistakes when they occur. It would be helpful, for example, when a semantic error occurs, that an appropriate window is opened, and the problem icons or links are visually highlighted.
5. *Provide visual support for execution* At a minimum however, state information could be displayed in the icon itself. A good use of this might be changing the appearance

of a switch icon to simulate its on or off configuration, or changing the color of an LED icon to simulate its being lit (in the digital circuits domain).

6. *Incorporation of a "mixed-mode" execution capability in the subsystems* The initial implementation of events into Architect allows only one mode of execution for a developed application. There are circumstances where we want some independent subsystems of an application to execute in a non-event-driven sequential mode while others execute in an event-driven mode.

7. *Concurrent simulation* The event-driven and time-driven simulation capabilities within Architect allow for only a single thread of control within an application. Expand Architect to include an event-driven and time-driven concurrent simulation capability with multiple threads of control. A decision will have to be made as to what level of nesting will be allowed for the concurrent processing of subsystems. The level of concurrency must determine whether only the top level independent subsystems execute concurrently, or whether subsystems subordinate to the top level subsystems execute concurrently.

6.4 *Final Comments*

The graphical user interface enhanced during this thesis effort is another step in improving software development techniques. Application specialists will be relying more on visualizing their applications. The emphasis on user interfaces will need to be more user centered and less implementer oriented. Keeping the user in the loop will result in a more usable visual user interface, thus ensuring improved application composition.

Domain-oriented application composition and generation systems like Architect will be more commonplace than ever before so, visual user interfaces will be even more important.

Appendix A. Sample Session for AVSI III

This appendix contains a sample session in which a BCD to Excess-3 Decoder is built from the primitive objects defined in the logic-circuits domain. The circuit diagram for the Decoder is shown in Figure A.1.

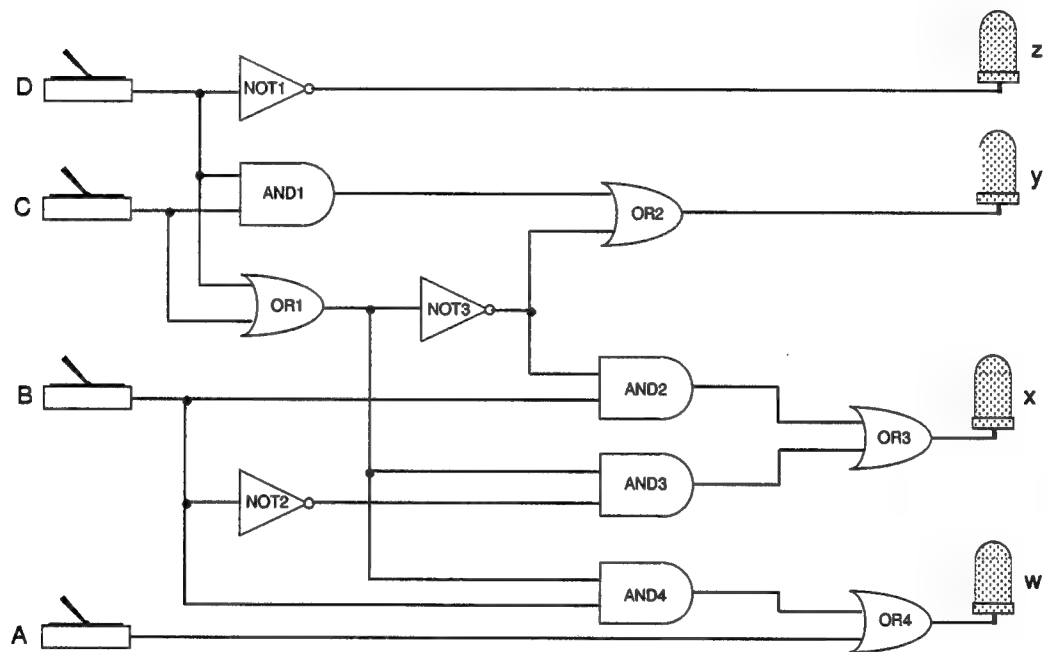


Figure A.1 BCD to Excess-3 Decoder Circuit

A.1 Starting AVSI

REFINE must be loaded in the Emacs window from the Architect directory. Once this is accomplished enter:

```
(load "1")
```

When the prompt returns, enter:

```
(1)
```

It will take several minutes for this file to run because it loads the DIALECT and INTERVISTA systems as well as the Architect and AVSI III files. When the load is complete, a prompt appears:

```
Load complete
Type "(AVSI)" to start AVSI
NIL
.>
```

Now enter the command:

```
(avsi)
```

This action loads the visual specification files for the domains currently defined for Architect. After the visual information is parsed into the object base, the control panel (shown in Figure 3.1) appears in the upper left-hand corner of the screen. Across the top of the window is a row of buttons that are used to invoke many of the application composition functions of AVSI III. The lower portion of the window is a message area used to display status and errors.

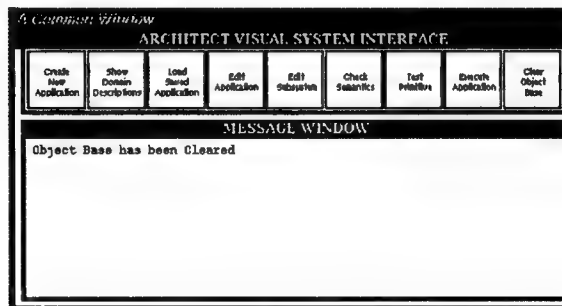


Figure A.2 AVSI III Control Panel

A.2 Create a New Application

To create a new application:

1. Click any mouse button on the button labeled **Create New Application**.
2. A pop-up window appears and prompts, **Select Domain**. Click on the menu item **CIRCUITS**.
3. A pop-up window appears with the prompt, **Enter name of application**. Type
bcd-xs3
4. The name can be entered by hitting the "return" key or by clicking on **Do It** at the bottom of the pop-up window.

A blue system-composition-window labeled, **System Composition Window** for the application **BCD-XS3** appears, containing a single icon labeled,

APPLICATION - OBJ
BCD - XS3

. A green window labeled, **Technology Base Window for CIRCUITS**, also appears and contains an icon for each primitive-object in the current domain (refer to Figure A.3).

A.3 Edit the Application

Now that the application has been created, the next step is to edit the application's elements. Editing an application is comprised of two separate operations: editing an application's components, and editing an application's update algorithm.

A.3.1 To add a controlling subsystem-obj to the application:

1. Click on the diagram surface (anywhere on the blue surface except within the icon's boundary) of the window. A pop-up menu will appear.

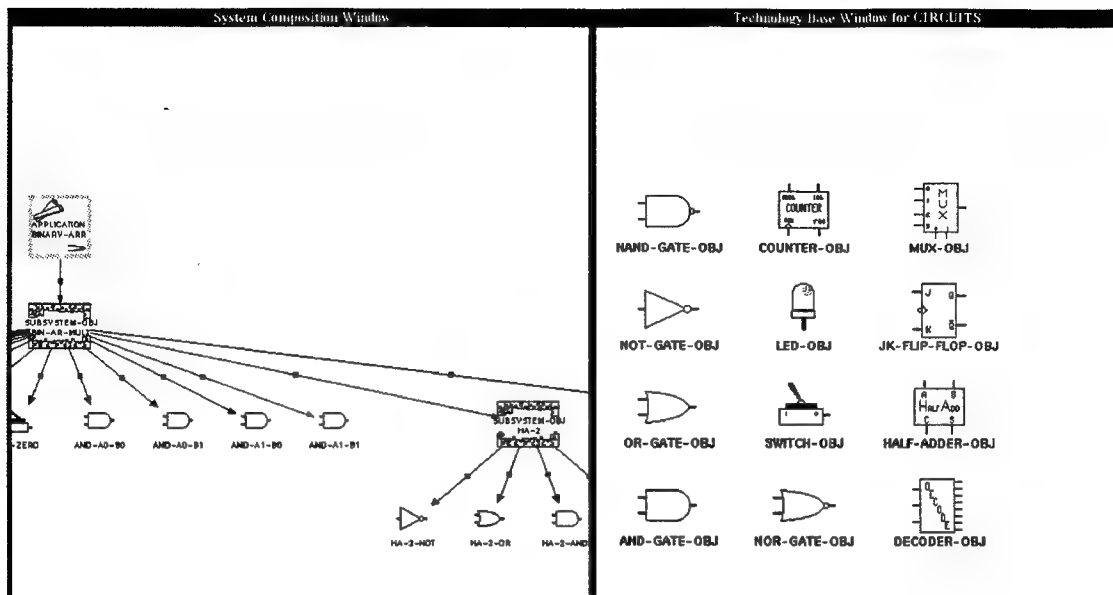


Figure A.3 System Composition Window and Technology Base Window

2. Select **Create New Subsystem**.
3. A pop-up window appears, with the prompt **Enter a name:**. Enter **driver**
4. A box outline of an icon appears, attached to the mouse cursor. Place the icon below the application-obj icon by moving the cursor to the desired location and clicking.
5. Click any mouse button on the newly created subsystem-obj icon and select the menu option **Link to Source**.
6. The mouse cursor changes from an arrow to an oval with a dot in it, signifying that an object needs to be selected. Place the mouse cursor on the application-obj's icon and click any mouse button. A link appears between the application-obj's icon and the subsystem-obj's icon, as in Figure A.5.

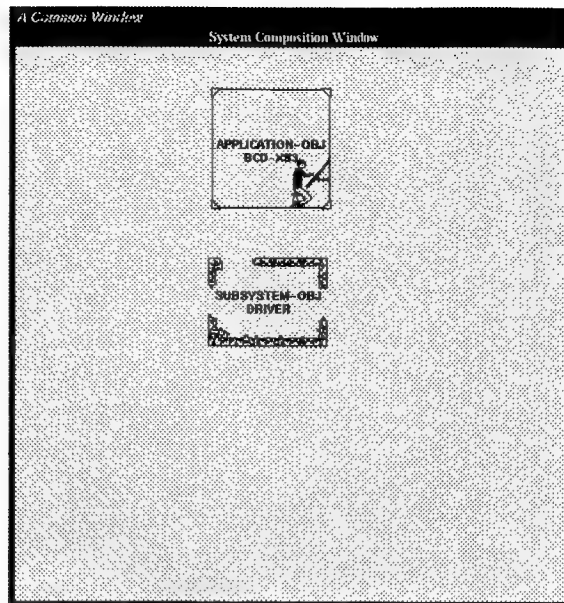


Figure A.4 System-Composition-Window

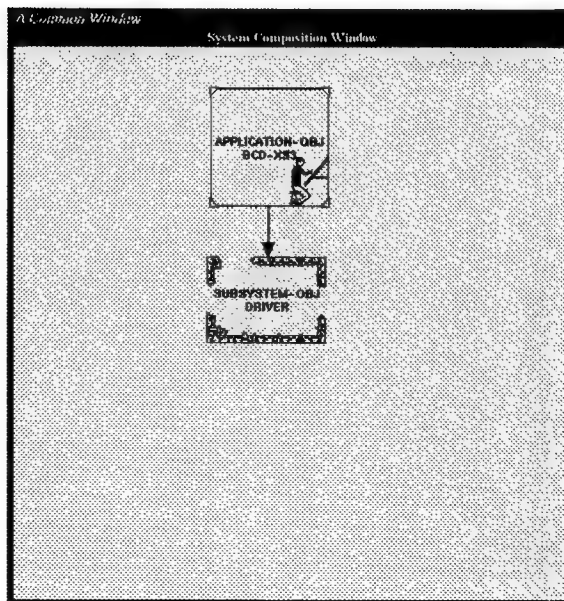


Figure A.5 System Composition-Window

A.4 Edit the Subsystems

Building a subsystem is similar to building the application. This section illustrates instantiating and linking primitive-objs and a nested subsystem-obj to the controlling subsystem created in the previous section.

The subsystem, **DRIVER**, will control four switches, four LEDS, and a separate subsystem (called bcd-excess3), which is the heart of the application. Bcd-excess3 consists of three not-gates, four and-gates, and four or-gates (refer back to Figure A.1). To add these objects, perform the following steps:

1. Click on the blue diagram surface and select the **Create New Subsystem** menu item.
2. Name the new subsystem **bcd-excess3**.
3. Place the subsystem-icon somewhere on the blue window.

A.4.1 To add the primitive objects:

1. Click on the icon in the green technology-base-window labeled **SWITCH-OBJ**, that looks like a toggle switch.
2. A Switch-icon is created and attached to the mouse cursor. Place this icon on in the blue edit-subsystem-window near **DRIVER**.
3. Name the switch by typing **A** in the pop-up window.
4. Follow the above steps to create and place three more switch objects, named **B**, **C**, and **D**.

5. Similarly, create four LED objects named **W**, **X**, **Y**, and **Z**.

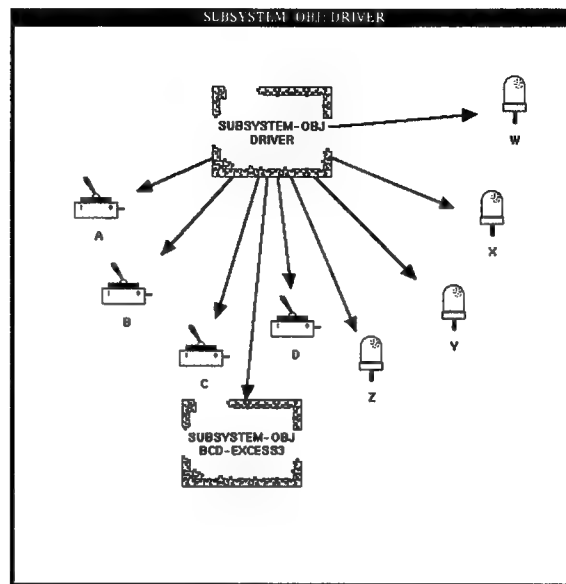


Figure A.6 Edit-Subsystem-Window

6. Link the primitive objects and **BCD-EXCESS3** to **DRIVER** by clicking a mouse button on **DRIVER**, and selecting **Link Multiple Targets** from the pop-up menu.
7. A pop-up window will appear that lists all the unconnected objects in the **System Composition Window**. Select **All of the Above**, and click on **Do It**. A link will appear from **DRIVER** to each of the other icons. Figure A.6 shows the **System Composition Window** at this point.
8. Now click on the **Technology Base Window** icon that represents a not-gate.
9. Position it near **BCD-EXCESS3** and click a mouse button to “drop” it.
10. Name the not-gate by typing **not1** in the pop-up window.
11. Repeat the process to add **not2** and **not3**.

12. Select the the appropriate icons from the **Technology Base Window** and create **and1, and2, and3, and4, or1, or2, or3, and or4**.
13. Link the new primitive objects to **BCD-EXCESS3** by clicking a mouse button on **BCD-EXCESS3**, and selecting **Link Multiple Targets** from the pop-up menu. Select **All of the Above**, and click on **Do It**.

Note: The window is probably cluttered and disorganized at this point. You can clean up the display by clicking on the blue background of the **System Composition Window** and selecting **Redraw**. The window is redrawn in an inverted tree-layout with **DRIVER** at the root and the other objects arranged underneath. The entire subsystem will not be visible in the window. You can scroll the window vertically or horizontally using the scroll-bars on the left and bottom of the window. The window can be resized by clicking the mouse on the black title bar and selecting **Reshape**. You can also change the size of the icons in the window by clicking on the blue surface and selecting **Change Scale Factor** from the menu. A new window will appear prompting **Full Size (1:1)** or **Half Size (1:2)**. Select half size and the window will be redrawn with the icons at half scale. You may, at any time, pretty-print an object (show its object base representation) by clicking on its icon and choosing the menu selection **Pretty-Print Object**.

A.4.2 To connect DRIVER's Imports and Exports: To connect the import and export objects perform the following steps:

1. Click a mouse button on the subsystem-obj icon labeled **DRIVER** in **BCD-XS3's System Composition Window**.

A red window, labeled **Imports/Exports** will open and contain the two subsystem-obj OCU-like icons representing the bf BCD-EXCESS3 subsystem and the **DRIVER** subsystem. Select **Make Internal Connections** from the pop-up window. The import-export-window will reopen and contain the four switch icons, the four led icons, and an OCU-like icon representing the **BCD-EXCESS3** subsystem (as shown in Figure A.7). The black bars (these bars are actually highlighted subicons attached to the primitive's icon) on the sides of the primitive icons indicate connections that need to be made.

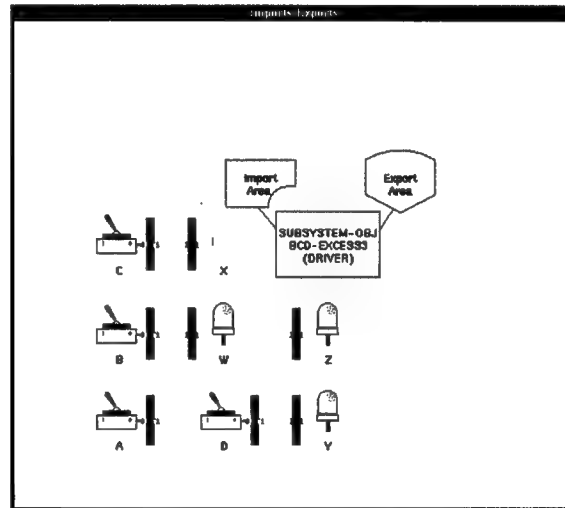


Figure A.7 **DRIVER's** Import-Export Window

2. Icons can be moved to new positions on the screen by clicking on the icon and selecting **Move Icon** from the pop-up menu. An square grid, attached to the mouse, appears. Move the grid to the new icon position and click to "drop" the icon. An example of the repositioned icons is shown in Figure A.8.

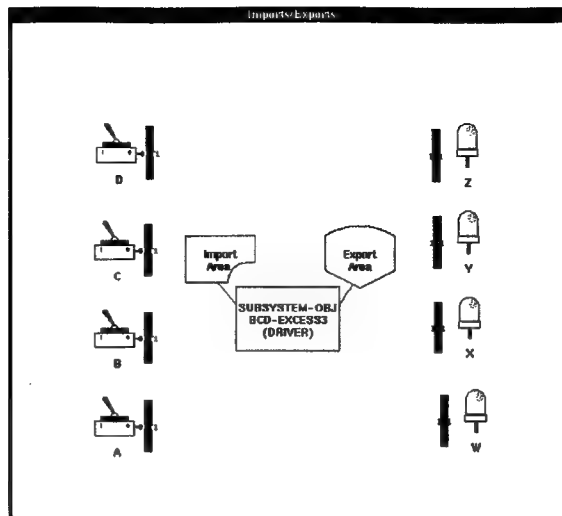


Figure A.8 Repositioned Icons

3. Click on the part of **BCD-EXCESS3**'s icon labeled **Import Area**. A white text window (a msp-window) will appear in the upper right area of the screen. The window lists the objects in **BCD-EXCESS3** that require input (see Figure A.9).

Import Area for BCD-EXCESS3			
Name	Category	Consumer	(Source: Obj, SS, Name)
IN1	SIGNAL	NOT1	
IN1	SIGNAL	NOT2	
IN1	SIGNAL	NOT3	
IN2	SIGNAL	AND1	
IN1	SIGNAL	AND1	
IN2	SIGNAL	AND2	
IN1	SIGNAL	AND2	
IN2	SIGNAL	AND3	
IN1	SIGNAL	AND3	
IN2	SIGNAL	AND4	
IN1	SIGNAL	AND4	
IN2	SIGNAL	OR1	
IN1	SIGNAL	OR1	
IN2	SIGNAL	OR2	
IN1	SIGNAL	OR2	
IN2	SIGNAL	OR3	
IN1	SIGNAL	OR3	
IN2	SIGNAL	OR4	
IN1	SIGNAL	OR4	

Figure A.9 Import Area for **BCD-EXCESS3**

4. Connect Switch **A** to gate **OR4** by clicking the mouse on the black bar on **A** and then clicking on the text entry **IN2 SIGNAL OR4** in the msp-window. Notice that

the black bar changes to a clear box (indicating the switch is connected to an object not visible on the screen), the msp-window is updated to reflect the connection, and another msp-window labeled "Export Area for DRIVER" appears. The export window shows the same connections being made from the switch's perspective.

5. In the same fashion, make the following connections:

switch B	connected to	IN1 NOT2
switch B	connected to	IN2 AND2
switch B	connected to	IN2 AND4
switch C	connected to	IN2 AND1
switch C	connected to	IN2 OR1
switch D	connected to	IN1 NOT1
switch D	connected to	IN1 AND1
switch D	connected to	IN1 OR1

In a similar manner, the leds must be connected.

6. Click on the part of **BCD-EXCESS3**'s icon labeled **Export Area**. A white text window (a msp-window) will appear in the lower right area of the screen. The window lists the objects in **BCD-EXCESS3** that provide output.

7. Connect Led **W** to gate **OR4** by clicking the mouse on the black bar on **W** and then clicking on the text entry **OUT1 SIGNAL OR4** in the msp-window. Again, the black bar changes to a clear box (indicating the led is connected to an object not visible on the screen), the msp-window is updated to reflect the connection, and another msp-window labeled "Import Area for DRIVER" appears. The import window shows the same connections being made from the led's perspective.

8. In the same fashion, make the following connections:

led X	connected to	OUT1 OR3
led Y	connected to	OUT1 OR2
led Z	connected to	OUT1 NOT1

When the connections have been made, the import-export-window should look something like Figure A.10 and **BCD-EXCESS3**'s msp-windows should look like Figure A.11.

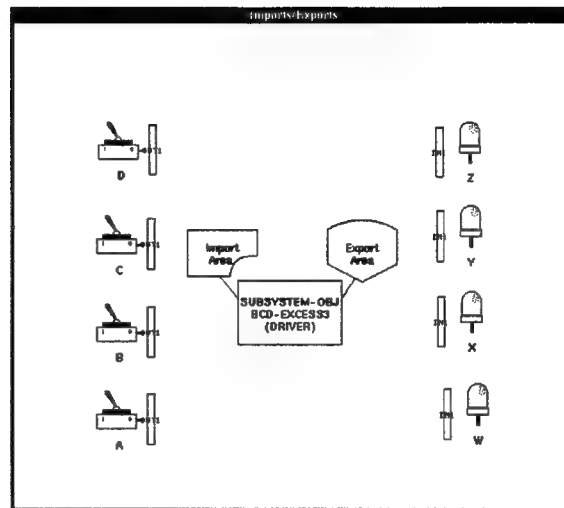


Figure A.10 **DRIVER**'s Import-Export Window

9. Close the import-export-window and the msp-windows by clicking on the red surface and selecting **Deactivate**, or by selecting **Deactivate** from each window's title bar menu. The **System Composition Window** is again displayed.

A.4.3 To create the application-obj's update-algorithm:

1. Click a mouse button on the application-obj icon **bcd-xs3**.
2. A pop-up menu appears prompting you to answer **YES** if you want to automatically build an update algorithm or **NO** if you want to build it manually. Three windows will appear (refer to Figure A.12). One contains a graphical view of the update algorithm, one contains a textual view of the algorithm, and the third (the Controllee Window) shows the icons that represent the application-obj's controllees

Import Area for BCD-EXCESS3			
Name	Catagory	Consumer	(Source: Obj, SS, Name)
IN1	SIGNAL	NOT1	(D, DRIVER, OUT1)
IN1	SIGNAL	NOT2	(B, DRIVER, OUT1)
IN1	SIGNAL	NOT3	
IN2	SIGNAL	AND1	(C, DRIVER, OUT1)
IN1	SIGNAL	AND1	(D, DRIVER, OUT1)
IN2	SIGNAL	AND2	(B, DRIVER, OUT1)
IN1	SIGNAL	AND2	
IN2	SIGNAL	AND3	
IN1	SIGNAL	AND3	
IN2	SIGNAL	AND4	(B, DRIVER, OUT1)
IN1	SIGNAL	AND4	
IN2	SIGNAL	OR1	(C, DRIVER, OUT1)
IN1	SIGNAL	OR1	(D, DRIVER, OUT1)
IN2	SIGNAL	OR2	
IN1	SIGNAL	OR2	
IN2	SIGNAL	OR3	
IN1	SIGNAL	OR3	
IN2	SIGNAL	OR4	(A, DRIVER, OUT1)
IN1	SIGNAL	OR4	

Export Area for BCD-EXCESS3			
Name	Catagory	Producer	(Target: Obj, SS, Name)
OUT1	SIGNAL	NOT1	(Z, DRIVER, IN1)
OUT1	SIGNAL	NOT2	
OUT1	SIGNAL	NOT3	
OUT1	SIGNAL	AND1	
OUT1	SIGNAL	AND2	
OUT1	SIGNAL	AND3	
OUT1	SIGNAL	AND4	
OUT1	SIGNAL	OR1	
OUT1	SIGNAL	OR2	(Y, DRIVER, IN1)
OUT1	SIGNAL	OR3	(X, DRIVER, IN1)
OUT1	SIGNAL	OR4	(W, DRIVER, IN1)

Figure A.11 Import-Export MSP-Windows

(with two extra icons for if-then and while-do constructs). The graphical update window contains two icons, "Start" and "End," with a dotted arrow pointing from the start-icon to the end-icon. If you choose to automatically build an update algorithm, the subsystem-obj labeled **DRIVER** will appear in the graphical update window between the "Start" and "End" icons.

3. If building the update algorithm manually follow the following steps:

- (a) Click a mouse button on the icon in the controllee window labeled **SUBSYSTEM - OBJ DRIVER**.

The cursor changes to an oval with a dot in it indicating that an object needs to be selected.

- (b) Click on the "nub" on the dotted line midway between the start and end icons.

This will cause the update sequence to redraw with the subsystem-obj included (see Figure A.13). Note the textual representation is automatically updated to reflect each change in the diagram window.

4. Close the edit-update-algorithm windows by clicking either on the black title bar at the top of the graphical update window, or in the green background of the window labeled, **Application Update Algorithm** and selecting **Deactivate** from the pop-up menu.

A.4.4 To build DRIVER's Update Algorithm: After the edit-update-algorithm windows have been closed, the blue **System Composition Window** and the green **Technology Base Window** will again be visible. Building the update algorithm

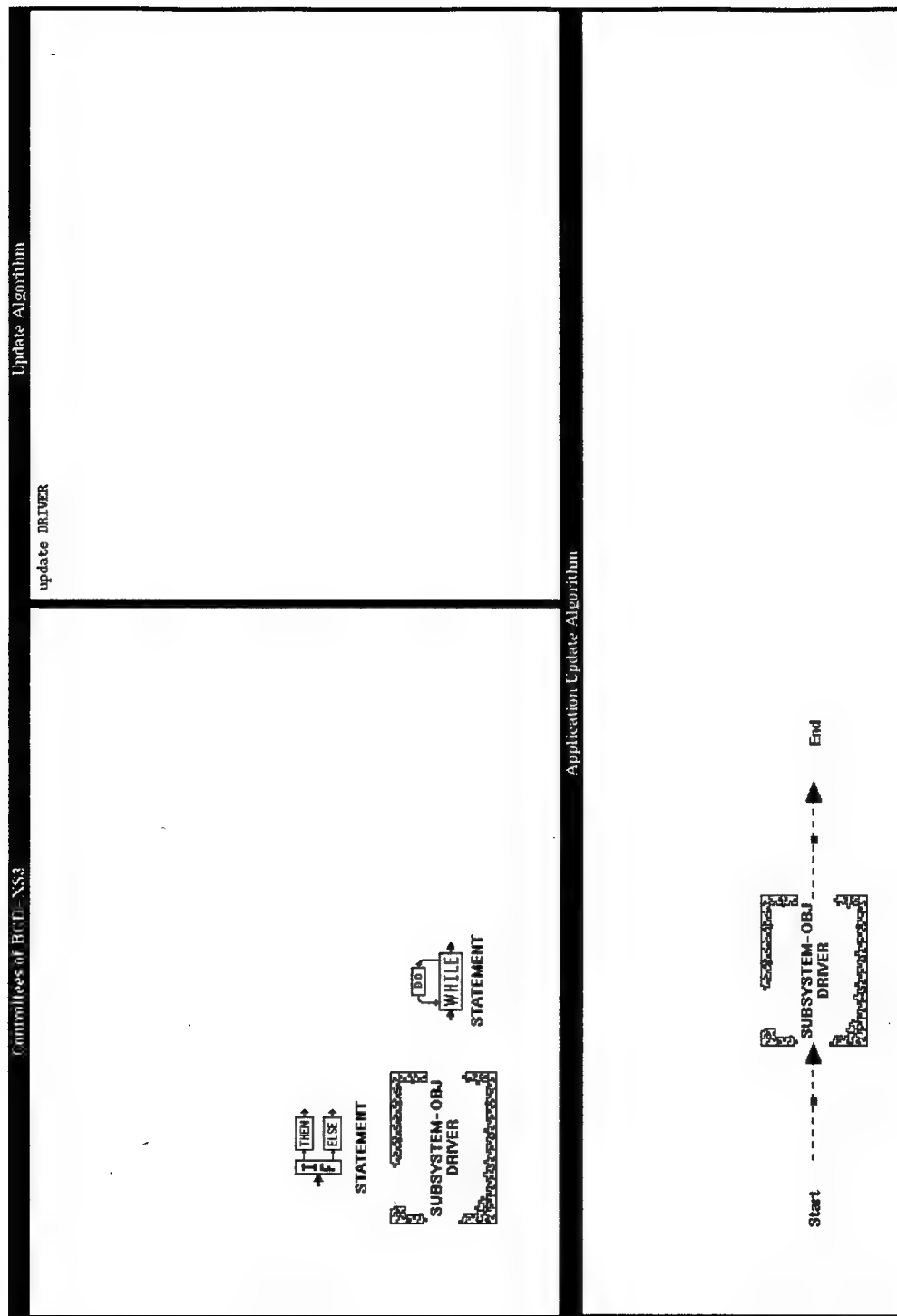


Figure A.12 Edit-Update-Algorithm Windows

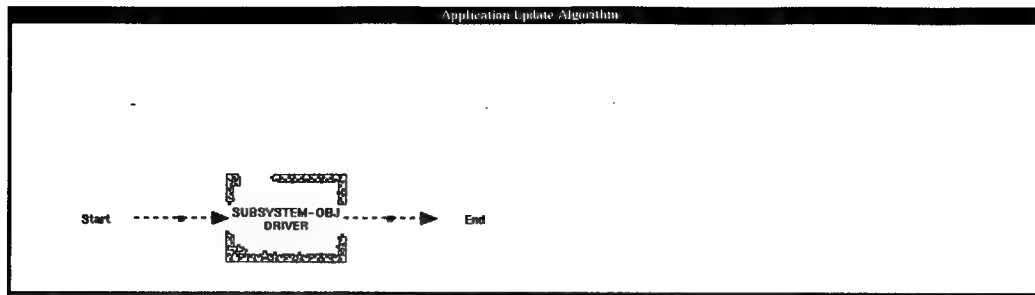


Figure A.13 Edit-Update-Algorithm

for **DRIVER** is similar to building the update algorithm for the application, and requires the following steps:

- (a) To build the update algorithm automatically:
- (b) A pop-up menu appears, answer **YES**. Three windows will appear (refer to Figure A.12). One contains a graphical view of the update algorithm, one contains a textual view of the algorithm, and the third (the Controllee Window) shows the icons that represent the subsystem-obj's controllees (with two extra icons for if-then and while-do constructs).
- (a) To build the update algorithm manually:
- (b) Click the mouse on the subsystem-obj icon labeled, **DRIVER** in the system composition window and select **Edit Update Algorithm**. The three windows that were seen in Figure A.12 are exposed, except the controllee window now contains the switches, leds, and subsystem controlled by **DRIVER**.
- (c) Add each controllee to the update sequence by clicking on the controllee icon and then clicking on the "nub" in the graphical update window that represents the

proper sequence position for the controllee. The order in which the controllees must appear is:

A B C D BCD-EXCESS3 W X Y Z

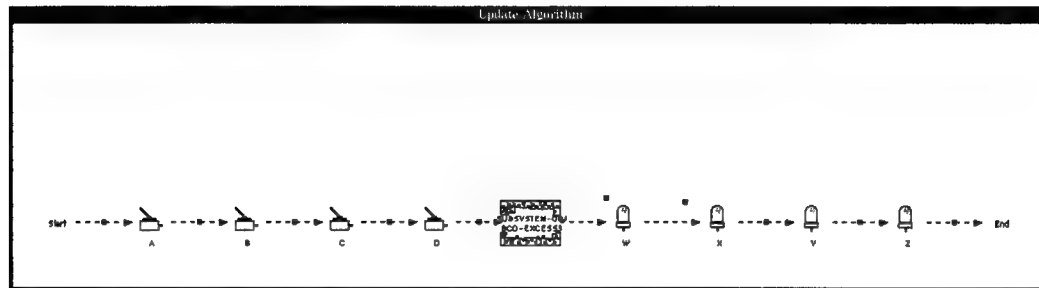


Figure A.14 **DRIVER's** Update Algorithm

When the fifth icon is added to the sequence, the window will automatically resize to keep the entire sequence visible.

Figure A.14 shows the completed update algorithm. Note that the textual update description is updated as the graphical update is built.

5. Close the windows by either clicking on the graphical update window title bar or clicking on the green background of the **Update Algorithm** and selecting **Deactivate** from the pop-up menu.

*A.4.5 To connect **BCD-EXCESS3's** Imports and Exports:* So far, the application has been created, the controlling subsystem (**DRIVER**) has been created, and **DRIVER's** components, import-export connections, and update algorithms have been defined. The definition process for **DRIVER** now needs to be repeated for **BCD-**

EXCESS3. Begin by defining **BCD-EXCESS3**'s import-export connections in the following way:

1. Click a mouse button on subsystem-obj icon labeled **BCD-EXCESS3**.
2. Select **Make Connections** from the pop-up window. An Imports/Exports window as in Figure A.15 will be displayed. As with the earlier import-export-window, the black bars (again, actually subicons) represent connections to be made, and the clear boxes (also subicons) represent connections made to objects not visible in the window.

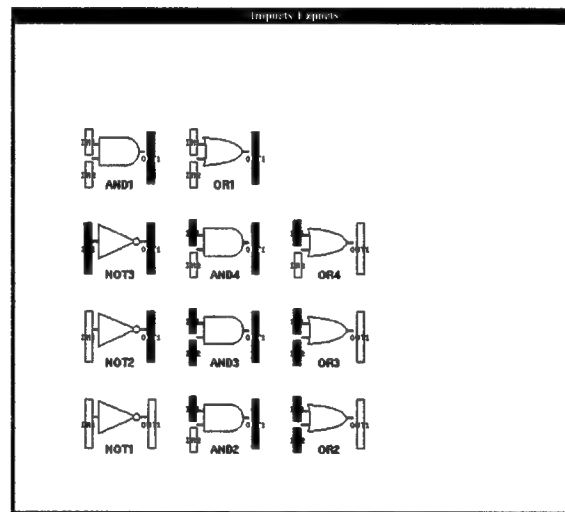


Figure A.15 **BCD-EXCESS3**'s Import-Export Window

3. Reposition the icons to resemble the circuit diagram in Figure A.1. The icons are repositioned by clicking on the icon, selecting **Move Icon** from the pop-up menu, moving the grid to the desired location, and clicking to "drop" the icon. The repositioned icons will probably look something like Figure A.16.

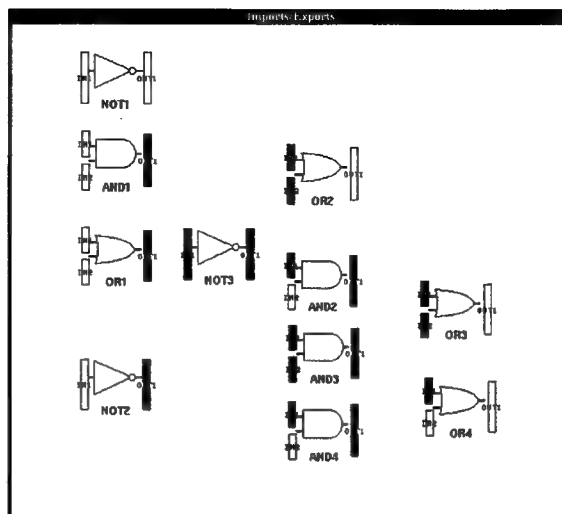


Figure A.16 BCD-EXCESS3's Import-Export Window

4. Connect **AND1** to **OR2** by clicking on the black bar labeled OUT1 on **AND1** and then clicking on the black bar labeled IN1 on **OR2**. Note that a line (a link) appears between the icons and the black bars disappear. The subicons of the primitives are still present, and are highlighted by a thin white line when the mouse is over one.
5. Make the remaining connections as described below:

AND1 OUT1	connected to	OR2 IN1
OR1 OUT1	connected to	NOT3 IN1
OR1 OUT1	connected to	AND3 IN1
OR1 OUT1	connected to	AND4 IN1
NOT2 OUT1	connected to	AND3 IN2
NOT3 OUT1	connected to	OR2 IN2
NOT3 OUT1	connected to	AND2 IN1
AND2 OUT1	connected to	OR3 IN1
AND3 OUT1	connected to	OR3 IN2
AND4 OUT1	connected to	OR4 IN1

Once the connections have been made, the import-export window should resemble Figure A.17. If desired, the links, once made, can be "hand-drawn" by clicking on the link's nub and selecting **Re-Draw Path**. The existing link will be deleted, the

mouse cursor will change to a “pencil,” and a dashed line will connect the icon to the cursor. Draw the link by moving the cursor to a new position and clicking the mouse button. To finish drawing the link, click the mouse button on a subicon. Also, if you find the labels on the subicons intrusive, they can be suppressed by clicking on the window surface and selecting **Clip Icon Labels** from the pop-up menu. Note: Although **NOT1** doesn’t appear to have any connections, it is actually connected to the export of **D** and the import of **Z**.

6. Close the import-export-window by clicking on the red surface and selecting **Deactivate** from the menu.

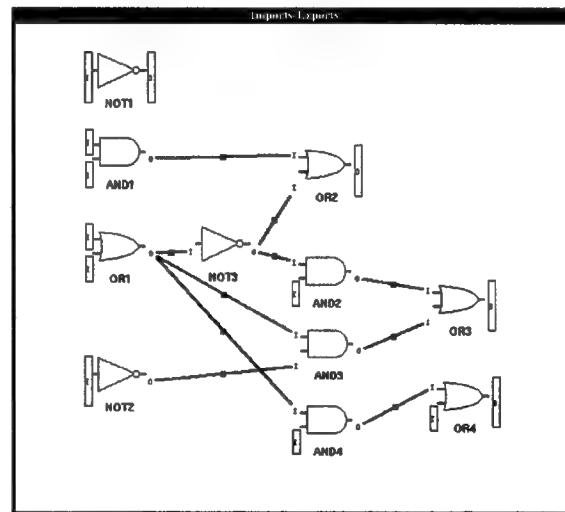


Figure A.17 **BCD-EXCESS3**'s Import-Export Window

*A.4.6 To build **BCD-EXCESS3**'s Update Algorithm:* When the import-export-window closes, the subsystem window will be visible. To build the update algorithm for **BCD-EXCESS3** do the following:

1. Click the mouse on the **SUBSYSTEM-OBJ** icon labeled, **BCD-EXCESS3** in the **System Composition Window**. The three update windows will be exposed, and the controllee window will contain the controllees defined for **BCD-EXCESS3**.
2. Create the update sequence by clicking the mouse on an icon in the controllee window and then clicking on the nub in the graphical update window as was done with the application and **DRIVER** update algorithms. Add the controllees to the update algorithm in the following sequence:

NOT1 AND1 OR1 NOT2 NOT3 OR2 AND2 AND3 AND4 OR3 OR4

The completed sequence is shown in Figure A.18.

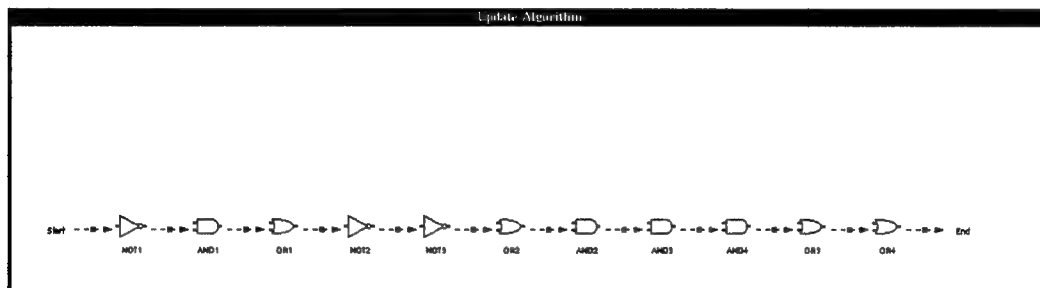


Figure A.18 **BCD-EXCESS3**'s Update Algorithm

3. Close the update windows by selecting **Deactivate** from the title bar menu of the graphical update window.

A.5 *Perform Semantic Checks*

Semantic checks are performed by Architect as part of the import-export connection process. However, the semantic checks may be run at any time by clicking on the control panel button labeled **Check Semantics** or clicking on any of the composition's icons and

selecting **Check Semantics**. The results of the semantic checks may be viewed in the EMACS window.

A.6 Execute the Application

Now that the application has been fully defined, it can be executed. The truth table for the BCD-XS3 decoder is shown in Table A.1.

Table A.1 BCD to Excess-3 Decoder Truth Table

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

The default position for a switch is “on,” however 1 1 1 1 is not a defined input for a BCD to Excess3 Decoder. Therefore, the switch attributes have to be set to a meaningful input in order to get a valid output from the execution. To set the switch attributes, do the following:

1. Click on the switch icon labeled **A** in the **System-Composition Window**.
2. Choose **View/Edit attributes** from the pop-up window. A window appears, listing the switch attributes for **A**.
3. Click on the attribute, **Position**. A pop-up window appears, listing the current value of the switch.
4. Enter a new value for the switch position by typing

`avsi::off`

(The “avsi::” prefix is the package name and is required for symbols. It is not required for other data types such as numbers and strings)

5. Change the values for any other switches in the same manner as above.

Figure A.19 shows the application.

Once a valid input state has been achieved, click on the control panel button labeled **Execute Application** or click on any icon in the **System Composition Window** and select **Execute Application** from the pop-up menu. The results are displayed in the EMACS window. A new set of switch positions can be established, and the application re-executed.

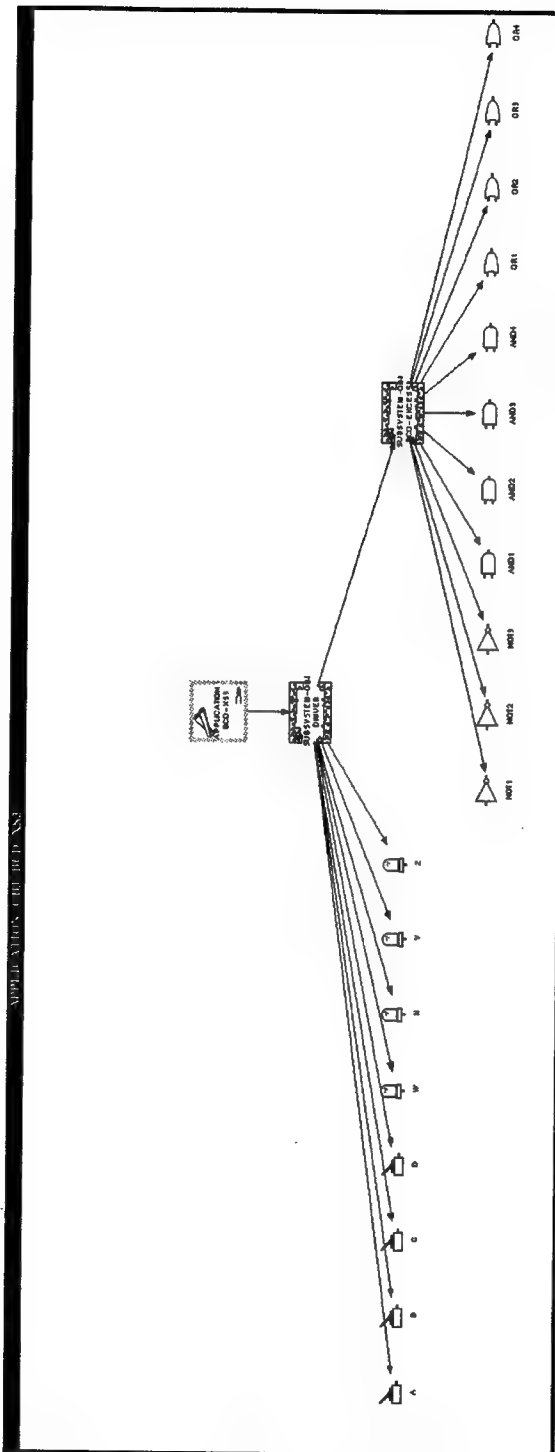


Figure A.19 BCD-XS3 Application

Appendix B. REFINETM Code Listings for AVSI III

This appendix contains a listing of the Lisp files required to run Architect and AVSI III. The order in which the files are listed below also indicates the required compilation order.

% Load system files for Dialect and Intervista

```
(load-system "dialect" "1-0")
(load-system "intervista" "1-0")
(load "AVSI-pkg.fas14")
```

% Load Architect files

```
(in-package 'AVSI)
(load "DSL/lisp-utilities.fas14")
(load "OCU-dm/dm-ocu.fas14")
(load "EXECUTIVE-TECH-BASE/dm-executive.fas14")
(load "DSL/globals.fas14")
(load "DSL/obj-utilities.fas14")
(load "DSL/read-utilities.fas14")
(load "DSL/menu.fas14")
(load "DSL/display-files.fas14")
(load "DSL/modify-obj")
(load "DSL/save.fas14")
(load "DSL/generic.fas14")
(load "DSL/build-generic.fas14")
(load "DSL/complete.fas14")
(load "DSL/set-globals.fas14")
(load "OCU/descriptor-tools.fas14")
(load "OCU/eval-expr.fas14")
(load "OCU/execute.fas14")
(load "OCU/imports-exports.fas14")
(load "OCU/semantic-checks.fas14")
```

% Load Executive Domain

```
(load "EXECUTIVE-TECH-BASE/var-executive.fas14")
(load "EXECUTIVE-TECH-BASE/ED-SEQ/ed-seq-event-man.fas14")
(load "EXECUTIVE-TECH-BASE/ED-SEQ/event-driven-clock.fas14")
```

```

(load "EXECUTIVE-TECH-BASE/ED-SEQ/connection-manager.fas14")
(load "EXECUTIVE-TECH-BASE/TD-SEQ/td-clock.fas14")
(load "EXECUTIVE-TECH-BASE/TD-SEQ/td-seq-event-man.fas14")

(load "OCU-dm/gram-ocu.fas14")
(load "lisp-file-utils.fas14")
(load "vsl-dm.fas14")
(load "vsl-gr.fas14")
(load "vsl-globals.fas14")
(load "vsl-utils.fas14")
(load "viz-utils.fas14")
(load "edit-expression.fas14")
(load "edit-update.fas14")
(load "edit-attr.fas14")
(load "create-obj.fas14")
(load "edit-ss.fas14")
(load "app-exe.fas14")
(load "edit-applic.fas14")
(load "tech-base.fas14")
(load "imp-exp.fas14")
(load "test-primitive.fas14")
(load "viz.fas14")

```

% Load Logic Circuits Domain

```

(load "CIRCUITS-TECH-BASE/dm-logic.fas14")
(load "CIRCUITS-TECH-BASE/var-circuits.fas14")
(load "CIRCUITS-TECH-BASE/or-gate.fas14")
(load "CIRCUITS-TECH-BASE/and-gate.fas14")
(load "CIRCUITS-TECH-BASE/nand-gate.fas14")
(load "CIRCUITS-TECH-BASE/nor-gate.fas14")
(load "CIRCUITS-TECH-BASE/not-gate.fas14")
(load "CIRCUITS-TECH-BASE/switch.fas14")
(load "CIRCUITS-TECH-BASE/jk-flip-flop.fas14")
(load "CIRCUITS-TECH-BASE/led.fas14")
(load "CIRCUITS-TECH-BASE/counter.fas14")
(load "CIRCUITS-TECH-BASE/decoder.fas14")
(load "CIRCUITS-TECH-BASE/half-adder.fas14")
(load "CIRCUITS-TECH-BASE/mux.fas14")
(load "CIRCUITS-TECH-BASE/gram-logic.fas14")

```

% Load EVENT-DRIVEN Logic Circuits Domain

```

(load "ED-CIRCUITS-TECH-BASE/dm-circuits.fas14")
(load "ED-CIRCUITS-TECH-BASE/var-ed-circuits.fas14")

```

```

(load "ED-CIRCUITS-TECH-BASE/or-gate.fas14")
(load "ED-CIRCUITS-TECH-BASE/and-gate.fas14")
(load "ED-CIRCUITS-TECH-BASE/nand-gate.fas14")
(load "ED-CIRCUITS-TECH-BASE/nor-gate.fas14")
(load "ED-CIRCUITS-TECH-BASE/not-gate.fas14")
(load "ED-CIRCUITS-TECH-BASE/switch.fas14")
(load "ED-CIRCUITS-TECH-BASE/jk-flip-flop.fas14")
(load "ED-CIRCUITS-TECH-BASE/led.fas14")
(load "ED-CIRCUITS-TECH-BASE/counter.fas14")
(load "ED-CIRCUITS-TECH-BASE/decoder.fas14")
(load "ED-CIRCUITS-TECH-BASE/half-adder.fas14")
(load "ED-CIRCUITS-TECH-BASE/mux.fas14")
(load "ED-CIRCUITS-TECH-BASE/one-shot.fas14")
(load "ED-CIRCUITS-TECH-BASE/clock.fas14")
(load "ED-CIRCUITS-TECH-BASE/gram-ed-logic.fas14")
(load "ED-CIRCUITS-TECH-BASE/ed-circuits-semantic-checks.fas14")

```

% Load TIME-DRIVEN Cruise-Missile Domain

```

(excl::run-shell-command "setenv KHOROS_HOME /apps/Khoros")
(load "CRUISE-MISSILE-TECH-BASE/dm-cm")
(load "CRUISE-MISSILE-TECH-BASE/airframe")
(load "CRUISE-MISSILE-TECH-BASE/autopilot")
(load "CRUISE-MISSILE-TECH-BASE/fuel-tank")
(load "CRUISE-MISSILE-TECH-BASE/guidance")
(load "CRUISE-MISSILE-TECH-BASE/jet-engine")
(load "CRUISE-MISSILE-TECH-BASE/navigation")
(load "CRUISE-MISSILE-TECH-BASE/throttle")
(load "CRUISE-MISSILE-TECH-BASE/warhead")
(load "CRUISE-MISSILE-TECH-BASE/gram-cm")
(load "CRUISE-MISSILE-TECH-BASE/var-cm.fas14")
(load "CRUISE-MISSILE-TECH-BASE/cm-semantic-checks.fas14")

```

% Load DSP Domain

```

(excl::run-shell-command "setenv KHOROS_HOME /apps/Khoros")
(load "DSP-TECH-BASE/dm-dsp.fas14")
(load "DSP-TECH-BASE/var-dsp.fas14")
(load "DSP-TECH-BASE/sinusoid.fas14")
(load "DSP-TECH-BASE/print-signal.fas14")
(load "DSP-TECH-BASE/graph-1-signal.fas14")
(load "DSP-TECH-BASE/graph-2-signal.fas14")
(load "DSP-TECH-BASE/graph-3-signal.fas14")
(load "DSP-TECH-BASE/graph-4-signal.fas14")
(load "DSP-TECH-BASE/save-signal.fas14")

```

```
(load "DSP-TECH-BASE/signal-adder.fas14")
(load "DSP-TECH-BASE/signal-multiplier.fas14")
(load "DSP-TECH-BASE/signal-subtractor.fas14")
(load "DSP-TECH-BASE/signal-divider.fas14")
(load "DSP-TECH-BASE/signal-abs-dif.fas14")
(load "DSP-TECH-BASE/dft.fas14")
(load "DSP-TECH-BASE/idft.fas14")
(load "DSP-TECH-BASE/scale-signal.fas14")
(load "DSP-TECH-BASE/unit-sample-sequence.fas14")
(load "DSP-TECH-BASE/unit-step-sequence.fas14")
(load "DSP-TECH-BASE/noise.fas14")
(load "DSP-TECH-BASE/piecewise-linear.fas14")
(load "DSP-TECH-BASE/stored-signal.fas14")
(load "DSP-TECH-BASE/adder.fas14")
(load "DSP-TECH-BASE/delay.fas14")
(load "DSP-TECH-BASE/multiplier.fas14")
(load "DSP-TECH-BASE/input-buffer.fas14")
(load "DSP-TECH-BASE/output-buffer.fas14")
(load "DSP-TECH-BASE/real-to-complex.fas14")
(load "DSP-TECH-BASE/complex-to-real.fas14")
(load "DSP-TECH-BASE/convolution.fas14")
(load "DSP-TECH-BASE/pad-signal.fas14")
(load "DSP-TECH-BASE/truncate-signal.fas14")
(load "DSP-TECH-BASE/reverse-signal.fas14")
(load "DSP-TECH-BASE/window-signal.fas14")
(load "DSP-TECH-BASE/time-filter.fas14")
(load "DSP-TECH-BASE/frequency-filter.fas14")
(load "DSP-TECH-BASE/user-designed-filter.fas14")
(load "DSP-TECH-BASE/gram-dsp.fas14")
(load "AVSI.fas14")
```

The REFINE source code for AVSI III may be obtained, upon request, from:

Maj Paul Bailor
AFIT/ENG
2950 P Street
Wright-Patterson AFB, OH 45433-7765

(513)255-6565 Ext. 4304
DSN 785-6565 Ext. 4304
email: pbailor@afit.af.mil

Bibliography

1. Anderson, Cynthia. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/92D-01, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.
2. Anderson Consulting, "Knowledge Base Software Architecture," 1994.
3. Cossentine, Jay A. *Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation System*. MS thesis, AFIT/GCS/ENG/93D-04, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.
4. Gool, Warren E. *Alternative Architectures for Domain-Oriented Application Composition and Generation Systems*. MS thesis, AFIT/GCS/ENG/93D-11, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.
5. Harris, Alfred W., Jr. *A Multiple Domain Capability For Domain-Oriented Application Composition Systems*. MS thesis, AFIT/GCS/ENG/94D-11, School of Engineering, Air Force Insitute of Technology (AU), Wright-Patterson AFB, OH, December 1994.
6. Hix, Deborah & H. Rex Hartson. *Developing User Interfaces, Ensuring Usability Through Product & Process*. Technical Report, John Wiley & Sons, Inc., 1993.
7. Jeffries, R., & Miller J.R., & Wharton, C., & Uyeda, K.M. "User Interface Evaluation in the Real World: A Comparison of Four Techniques." *In Proceedings of CHI'91*. 119-124. New York: ACM, 1991.
8. Kang, Kyo C. and others. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report (AD-A235785), Software Engineering Institute, November 1990.
9. Lee, Kenneth J. and others. *Model-Based Software Development*. Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.
10. Lewis, C., & Polson, O., & Wharton, C., & Rieman, J. "Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces." *In Proceedings of CHI'90*. 235-242. New York: ACM, 1990.
11. Marcus, Aaron. "Human communications issues in advanced UIs.," *Communication of the ACM*, 36(4) (April 1993).
12. Miller, G.A. "The Magical Number Seven, Plus or Minus Two: Some limits on our capability for procesgin information," *Psychological Science*, 63:81-97 (1956).
13. Myers, Brad A. *Why are Human-Computer Interfaces Difficult to Design and Implement?*. Technical Report CMU-CS-93-183, Computer Science Department, Carnegie Mellon University Pittsburgh, PA 15213: Carnegie Mellon University, July 1993.
14. Nielsen, J., & Molich, R. "Heuristic evaluation of user interfaces." *In Proceedings of CHI'90*. 249-256. New York: ACM, 1990.

15. Quest Windows Corporation, 5200 Great America Parkway, Santa Clara, CA 95054. *OSF/MotifTM Style Guide* (January 1992 Edition), January 1992.
16. Randour, Mary Anne. *Creating and Manipulating a Domain-Specific Formal Object Base to Support a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/92D-13, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.
17. Reasoning Systems Inc. *INTERVISTATM User's Guide*. Palo Alto, CA, 1991. For INTERVISTATM Version 1.0.
18. Shneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction, Second Edition*. New York: Addison-Wesley Publishing Co., 1992.
19. Smith, Sidney L. & Mosier, Jane N. *Guidelines for Designing User Interface Software*. Technical Report ESD-TR-86-278, Electronic Systems Division, AFSC Hanscom AFB, MA 01731-5000: The MITRE Corporation, August 1986.
20. Tognazzini, Bruce. *Tog on Interface*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1992.
21. Waggoner, Robert W. *Domain Modeling of Time-Dependent Systems*. MS thesis, AFIT/GCS/ENG/93D-23, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.
22. Warner, Russel M. *A Method for Populating the Knowledge Base of AFIT's Domain Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93D-24, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.
23. Weide, Timothy. *Development of a Visual System Interface to Support a Domain-oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93M-05, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.
24. Welgan, Robert L. *Domain Analysis and Modeling of a Model-Based Software Executive*. MS thesis, AFIT/GCS/ENG/93D-25, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

Vita

Captain Richard Anthony Guinto was born 5 May 1960 in Honolulu, Hawaii and graduated from Pearl City High School in Pearl City, Hawaii in June, 1978. He enlisted in the Air Force in March 1980 and completed technical training for general accounting at Shephard AFB, Texas in September 1980. He completed technical training for computer programming at Keesler AFB, Mississippi in December 1983. He spent twenty months as an IBM Mainframe Hardware Configuration Manager at Strategic Air Command Headquarters at Offutt AFB, Nebraska. By way of the Airmen Educational Commissioning Program, he was awarded a Bachelor's Degree in Computer Science at New Mexico State University in December 1988 and commissioned in the United States Air Force through the Air Force Officer Training School in May 1989. After attending the Communications-Computer Systems Officer School at Keesler AFB, Mississippi from May to August 1989, he served as a training systems software evaluator with the 3907th Systems Evaluation Squadron and a training systems test director with the Detachment 1, 31st Test and Evaluation Squadron at Castle AFB, California until July 1992. He was then assigned as the Small Computer Acquisition Manager for Air Force Materiel Command Headquarters at Wright-Patterson AFB, Ohio from July 1992 to May 1993. In May, 1993, he entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio to pursue a Master of Science degree in Computer Science. Upon graduation, Captain Guinto will be assigned to the Low Altitude Night Terrain InfraRed Navigation System Program Office at the Aeronautical System Center, Wright-Patterson AFB, Ohio.

Permanent address: 1132-C Hoola Place
Pearl City, HI 96782

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis		
4. TITLE AND SUBTITLE ENHANCED VISUAL USER INTERFACE SUPPORT FOR DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEMS			5. FUNDING NUMBERS	
5. AUTHOR(S) Richard A. Guinto				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/94D-06	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Rick Painter 2241 Avionics Circle, Suite 16 WL/AAWA-1 BLD 620 Wright-Patterson AFB, OH 45433-7765			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Distribution Unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) <p>This research refined the functionality and usability of a previously developed visual interface for a domain-oriented application composition system. The refinements incorporated more sophisticated user interface design concepts to reduce user workload. User workload was reduced through window reordering, menu redesign, and Human Computer Interaction techniques such as; combining repetitive procedures into single commands, reusing composition information whenever possible and deriving new information from existing information. The Software Refinery environment, including its visual interface tool INTERVISTA, was used to develop techniques for visualizing and manipulating objects contained in a formal knowledge base of objects. The interface was formally validated with digital logic-circuits, digital signal processing, event-driven logic-circuits, and cruise-missile domains. A comparative analysis of the application composition process with the previous visual interface was conducted to quantify the workload reduction realized by the new interface. Level of effort was measured as the number of user interactions (mouse or keyboard) required to compose an application. On average, application composition effort was reduced 34.0% for the test cases.</p>				
14. SUBJECT TERMS Human Computer Interface, Interfaces, Software Engineering, Computer Graphics			15. NUMBER OF PAGES 120	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet optical scanning requirements.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must be at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	PR - Project
G - Grant	TA - Task
PE - Program Element	WU - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Denote any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.3.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.